Refactoring: A First Example

Martin Fowler’s First Example of Refactoring, Adapted to Java

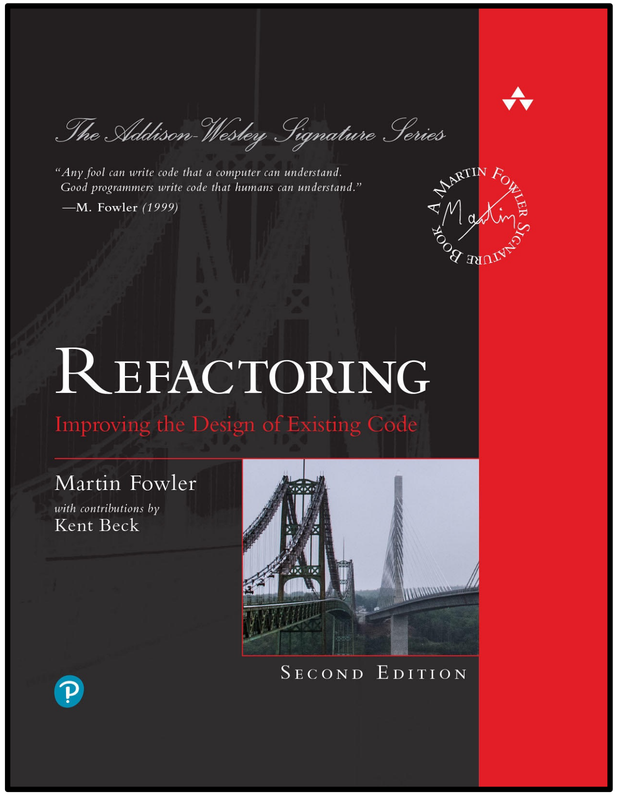
follow in the footsteps of **refactoring guru Martin Fowler**

as he **improves** the **design** of a program in a simple yet **instructive refactoring example**

whose **JavaScript** code and associated **refactoring** is herein adapted to **Java**

based on the second edition of ‘the’ **Refactoring** book

Martin Fowler
@martinfowler



The Addison-Wesley Signature Series

“Any fool can write code that a computer can understand. Good programmers write code that humans can understand.”
—M. Fowler (1999)

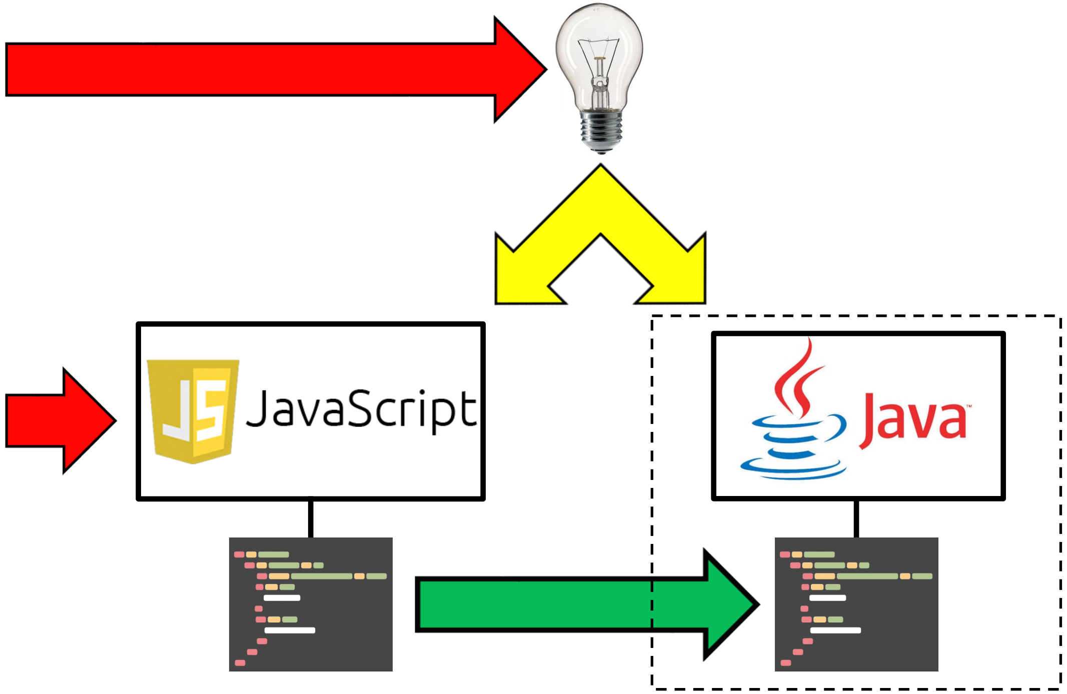
A Martin Fowler Signature Book

REFACTORING

Improving the Design of Existing Code

Martin Fowler
with contributions by
Kent Beck

SECOND EDITION



slides by

@philip_schwarz slideshare https://www.slideshare.net/pjschwarz

@philip_schwarz

Neither **Martin Fowler** nor the **Refactoring** book need any introduction.

I have always been a great fan of both, and having finally found the time to study in detail the **refactoring example** in the **second edition** of the book, I would like to share the experience of adapting to **Java** such a useful **example**, which happens to be written in **JavaScript**.

Another reason for looking in detail at the **example** is that it can be used as a good **refactoring code kata**.

While we'll be closely following **Martin Fowler**'s footsteps as he works through the **refactoring example**, and while those of you who don't already own a copy of the book will no doubt learn a lot about the chapter containing the **example**, what we'll see is obviously only a small part of what makes the book such a must have for anyone interested in **refactoring**.

The next four slides consist of excerpts in which **Martin Fowler** introduces the program whose **design** he will be **improving** through **refactoring**.

Martin Fowler
@martinfowler

So I’m going to start this book with an example of refactoring. I’ll talk about how refactoring works and will give you a sense of the refactoring process. I can then do the usual principles-style introduction in the next chapter.

With any introductory example, however, I run into a problem. If I pick a large program, describing it and how it is refactored is too complicated for a mortal reader to work through. (I tried this with the original book—and ended up throwing away two examples, which were still pretty small but took over a hundred pages each to describe.) **However, if I pick a program that is small enough to be comprehensible, refactoring does not look like it is worthwhile**.

I’m thus in the classic bind of anyone who wants to describe techniques that are useful for real-world programs.

Frankly, it is not worth the effort to do all the refactoring that I’m going to show you on the small program I will be using.

But if the code I’m showing you is part of a larger system, then the refactoring becomes important. Just look at my example and imagine it in the context of a much larger system.

JavaScript

I chose JavaScript to illustrate these refactorings, as I felt that this language would be readable by the most amount of people.

You shouldn’t find it difficult, however, to adapt the refactorings to whatever language you are currently using.

I try not to use any of the more complicated bits of the language, so you should be able to follow the refactorings with only a cursory knowledge of JavaScript.

My use of JavaScript is certainly not an endorsement of the language.

Although I use JavaScript for my examples, that doesn’t mean the techniques in this book are confined to JavaScript.

The first edition of this book used Java, and many programmers found it useful even though they never wrote a single Java class.

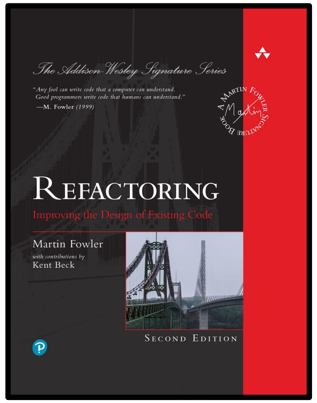
I did toy with illustrating this generality by using a dozen different languages for the examples, but I felt that would be too confusing for the reader.

Still, this book is written for programmers in any language.

Outside of the example sections, I’m not making any assumptions about the language.

I expect the reader to absorb my general comments and apply them to the language they are using.

Indeed, I expect readers to take the JavaScript examples and adapt them to their language.


The Addison-Wesley Signature Series
“Any fool can write code that a computer can understand. Good programmers write code that humans can understand.”
—M. Fowler (1999)
A Martin Fowler Signature Book
Refactoring
Improving the Design of Existing Code
Martin Fowler
with contributions by
Kent Beck
Second Edition

Martin Fowler

@martinfowler

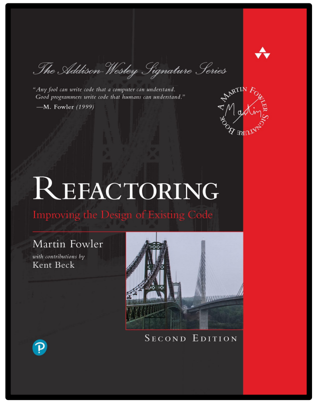

The Addison-Wesley Signature Series
"Any fool can write code that a computer can understand. Good programmers write code that humans can understand."
—M. Fowler (1999)
A Martin Fowler Signature Book
Martin
Refactoring
Improving the Design of Existing Code
Martin Fowler
with contributions by
Kent Beck
Second Edition

Image a company of theatrical players who go out to various events performing plays.

Typically, a customer will request a few plays and the company charges them based on the size of the audience and the kind of play they perform.

There are currently two kinds of plays that the company performs: tragedies and comedies.

As well as providing a bill for the performance, the company gives its customers "volume credits" which they can use for discounts on future performances—think of it as a customer loyalty mechanism.

The performers store data about their **plays** in a simple **JSON** file that looks something like this:

```
plays.json…

   {
     "hamlet": {"name": "Hamlet", "type": "tragedy"},
     "as-like": {"name": "As You Like It", "type": "comedy"},
     "othello": {"name": "Othello", "type": "tragedy"}
   }
```

The data for their **bills** also comes in a **JSON** file:

```
invoices.json…

  [
    {
      "customer": "BigCo",
      "performances": [
        {
          "playID": "hamlet",
          "audience": 55
        },
        {
          "playID": "as-like",
          "audience": 35
        },
        {
          "playID": "othello",
          "audience": 40
        }
      ]
    }
  ]
```

```
function statement (invoice, plays) {
  let totalAmount = 0;
  let volumeCredits = 0;
  let result = `Statement for ${invoice.customer}\n`;
  const format = new Intl.NumberFormat("en-US",
    { style: "currency", currency: "USD", minimumFractionDigits: 2 }).format;

  for (let perf of invoice.performances) {
    const play = plays[perf.playID];
    let thisAmount = 0;

    switch (play.type) {

      case "tragedy":
        thisAmount = 40000;
        if (perf.audience > 30)
          thisAmount += 1000 * (perf.audience - 30);
        break;

      case "comedy":
        thisAmount = 30000;
        if (perf.audience > 20)
          thisAmount += 10000 + 500 * (perf.audience - 20);
        thisAmount += 300 * perf.audience;
        break;

      default:
        throw new Error(`unknown type: ${play.type}`);
    }

    // add volume credits
    volumeCredits += Math.max(perf.audience - 30, 0);
    // add extra credit for every ten comedy attendees
    if ("comedy" === play.type) volumeCredits += Math.floor(perf.audience / 5);

    // print line for this order
    result += `  ${play.name}: ${format(thisAmount/100)} (${perf.audience} seats)\n`;
    totalAmount += thisAmount;
  }

  result += `Amount owed is ${format(totalAmount/100)}\n`;
  result += `You earned ${volumeCredits} credits\n`;
  return result;
}
```

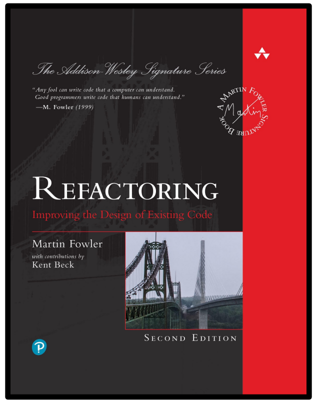
The code that prints the bill is this simple function.

What are your thoughts on the design of this program? The first thing I'd say is that it's tolerable as it is—a program so short doesn't require any deep structure to be comprehensible. But remember my earlier point that I have to keep examples small. Imagine this program on a larger scale—perhaps hundreds of lines long. At that size, a single inline function is hard to understand.

Given that the program works, isn't any statement about its structure merely an aesthetic judgment, a dislike of "ugly" code? After all, the compiler doesn't care whether the code is ugly or clean. But when I change the system, there is a human involved, and humans do care. A poorly designed system is hard to change—because it is difficult to figure out what to change and how these changes will interact with the existing code to get the behavior I want. And if it is hard to figure out what to change, there is a good chance that I will make mistakes and introduce bugs.

Thus, if I'm faced with modifying a program with hundreds of lines of code, I'd rather it be structured into a set of functions and other program elements that allow me to understand more easily what the program is doing. If the program lacks structure, it's usually easier for me to add structure to the program first, and then make the change I need.

Martin Fowler
@martinfowler



The Addison-Wesley Signature Series
REFACTORING
Improving the Design of Existing Code
Martin Fowler
with contributions by Kent Beck
SECOND EDITION

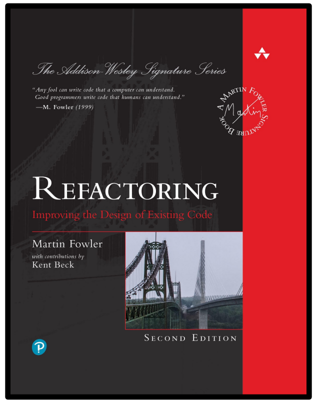
```
function statement (invoice, plays) {
  let totalAmount = 0;
  let volumeCredits = 0;
  let result = `Statement for ${invoice.customer}\n`;
  const format = new Intl.NumberFormat("en-US",
    { style: "currency", currency: "USD", minimumFractionDigits: 2 }).format;

  for (let perf of invoice.performances) {
    const play = plays[perf.playID];
    let thisAmount = 0;

    switch (play.type) {

      case "tragedy":
        thisAmount = 40000;
        if (perf.audience > 30)
          thisAmount += 1000 * (perf.audience - 30);
        break;

      case "comedy":
        thisAmount = 30000;
        if (perf.audience > 20)
          thisAmount += 10000 + 500 * (perf.audience - 20);
        thisAmount += 300 * perf.audience;
        break;

      default:
        throw new Error(`unknown type: ${play.type}`);
    }

    // add volume credits
    volumeCredits += Math.max(perf.audience - 30, 0);
    // add extra credit for every ten comedy attendees
    if ("comedy" === play.type) volumeCredits += Math.floor(perf.audience / 5);

    // print line for this order
    result += `  ${play.name}: ${format(thisAmount/100)} (${perf.audience} seats)\n`;
    totalAmount += thisAmount;
  }

  result += `Amount owed is ${format(totalAmount/100)}\n`;
  result += `You earned ${volumeCredits} credits\n`;
  return result;
}
```

In this case, I have a couple of changes that the users would like to make. First, they want a statement printed in **HTML**. Consider what impact this change would have. I'm faced with adding conditional statements around every statement that adds a string to the result. That will add a host of complexity to the function. Faced with that, most people prefer to copy the method and change it to emit **HTML**. Making a copy may not seem too onerous a task, but it sets up all sorts of problems for the future. Any changes to the charging logic would force me to update both methods—and to ensure they are updated consistently. **If I'm writing a program that will never change again, this kind of copy-and-paste is fine. But if it's a long-lived program, then duplication is a menace.**

This brings me to a second change. The players are looking to perform more kinds of plays: they hope to add **history**, **pastoral**, **pastoral-comical**, **historical-pastoral**, **tragical-historical**, **tragical-comical-historical-pastoral**, **scene individable**, and **poem unlimited** to their repertoire. They haven't exactly decided yet what they want to do and when. **This change will affect both the way their plays are charged for and the way volume credits are calculated.** As an experienced developer I can be sure that whatever scheme they come up with, they will change it again within six months. After all, when feature requests come, they come not as single spies but in battalions.

Martin Fowler

@martinfowler


The Addison-Wesley Signature Series
REFACTORING
Improving the Design of Existing Code
Martin Fowler
with contributions by Kent Beck
SECOND EDITION

@philip_schwarz

In this slide deck we are going to

1. Translate **Martin Fowler**'s initial **Javascript** program into **Java**
2. Follow in his **refactoring** footsteps, transforming our **Java** program so that it is **easier** to **understand** and **easier** to **change**.

On the very few occasions when a decision is made that turns out not to be a good fit in a **Java** context, we'll make an alternative decision that is more suitable for the **Java** version of the program.

In the process, we'll be using the following **Java** language features incorporated into **long-term support** (LTS) version **JDK 17** (the previous LTS version being **JDK 11**):

- **Text blocks** (**JDK 15**)
- **Records** (**JDK 16**)
- **Sealed interfaces** (**JDK 17**)

To keep the pace snappy, we'll sometimes coalesce a few of **Martin**'s **refactoring nanosteps** or **microsteps** into one (see next slide for a definition of these two types of **refactoring** step).

J. B. Rainsberger
@jbrains

Some Helpful Terms

In my lexicon, **a *nanostep* is something like adding a new field to a class**. Another **nanostep** is finding code that wrote to an existing field and adding code that writes the corresponding value to the new field, keeping their values synchronized with each other. Yet another is remembering the keystroke for "extract variable" so that you can simply type the expression (right-hand value) that you have in mind first, then assign it to a new variable (and let the computer compute the type of the variable for you).

A *microstep* is a collection of related nanosteps like introducing an interface *and* changing a few classes to implement that interface, adding empty/default method implementations to the classes that now need it. Another is pushing a value up out of the constructor into its parameter list. Yet another is remembering that you can either extract a value to a variable before extracting code into a method or you can extract the method first, then introduce the value as a parameter, and which keystrokes in NetBeans make that happen.

A *move* is a collection of related microsteps, like inverting the dependency between A and B, where A used to invoke B, but now A fires an event which B subscribes to and handles.

https://blog.thecodewhisperer.com/permalink/breaking-through-your-refactoring-rut

Let’s knock up some **Java** data structures for **plays**, **invoices** and **performances**.

```
plays.json…
  {
    "hamlet": {"name": "Hamlet", "type": "tragedy"},
    "as-like": {"name": "As You Like It", "type": "comedy"},
    "othello": {"name": "Othello", "type": "tragedy"}
  }
```

```
static final Map<String,Play> plays = Map.of(
  "hamlet" , new Play("Hamlet", "tragedy"),
  "as-like", new Play("As You Like It", "comedy"),
  "othello", new Play("Othello", "tragedy"));
```

```
record Play(String name, String type) { }
```

```
record Invoice(String customer, List<Performance> performances) { }
```

```
record Performance(String playID, int audience) { }
```

```
invoices.json…
  [
    {
      "customer": "BigCo",
      "performances": [
        {
          "playID": "hamlet",
          "audience": 55
        },
        {
          "playID": "as-like",
          "audience": 35
        },
        {
          "playID": "othello",
          "audience": 40
        }
      ]
    }
  ]
```

```
static final List<Invoice> invoices =
  List.of(
    new Invoice(
      "BigCo",
      List.of(new Performance( "hamlet", 55),
              new Performance("as-like", 35),
              new Performance("othello", 40))));
```

```javascript
function statement (invoice, plays) {
  let totalAmount = 0;
  let volumeCredits = 0;
  let result = `Statement for ${invoice.customer}\n`;
  const format = new Intl.NumberFormat("en-US",
    { style: "currency", currency: "USD", minimumFractionDigits: 2 }).format;

  for (let perf of invoice.performances) {
    const play = plays[perf.playID];
    let thisAmount = 0;

    switch (play.type) {

      case "tragedy":
        thisAmount = 40000;
        if (perf.audience > 30)
          thisAmount += 1000 * (perf.audience - 30);
        break;

      case "comedy":
        thisAmount = 30000;
        if (perf.audience > 20)
          thisAmount += 10000 + 500 * (perf.audience - 20);
        thisAmount += 300 * perf.audience;
        break;

      default:
        throw new Error(`unknown type: ${play.type}`);
    }

    // add volume credits
    volumeCredits += Math.max(perf.audience - 30, 0);
    // add extra credit for every ten comedy attendees
    if ("comedy" === play.type) volumeCredits += Math.floor(perf.audience / 5);

    // print line for this order
    result += `  ${play.name}: ${format(thisAmount/100)} (${perf.audience} seats)\n`;
    totalAmount += thisAmount;
  }

  result += `Amount owed is ${format(totalAmount/100)}\n`;
  result += `You earned ${volumeCredits} credits\n`;
  return result;
}
```

JavaScript

```java
static String statement(Invoice invoice, Map<String, Play> plays) {
  var totalAmount = 0;
  var volumeCredits = 0;
  var result = "Statement for " + invoice.customer() + "\n";
  final var formatter = NumberFormat.getCurrencyInstance(Locale.US);
  formatter.setCurrency(Currency.getInstance(Locale.US));

  for(Performance perf : invoice.performances()) {
    final var play = plays.get(perf.playID());
    var thisAmount = 0;

    switch (play.type()) {

      case "tragedy" -> {
        thisAmount = 40_000;
        if (perf.audience() > 30)
          thisAmount +=1_000 * (perf.audience() - 30);
      }

      case "comedy" -> {
        thisAmount = 30_000;
          if (perf.audience() > 20)
            thisAmount += 10_000 + 500 * (perf.audience() - 20);
          thisAmount += 300 * perf.audience();
      }

      default ->
        throw new IllegalArgumentException("unknown type " + play.type());
    }

    // add volume credits
    volumeCredits += Math.max(perf.audience() - 30, 0);
    // add extra credit for every ten comedy attendees
    if ("comedy" == play.type())
      volumeCredits += Math.floor(perf.audience() / 5);

    // print line for this order
    result += "  " + play.name() + ": " + formatter.format(thisAmount/100)
                   + " (" + perf.audience() + " seats)\n";
    totalAmount += thisAmount;
  }
  result += "Amount owed is " + formatter.format(totalAmount/100) + "\n";
  result += "You earned " + volumeCredits + " credits\n";
  return result;
}
```

Java

Here is a literal translation of the **Javascript** program into **Java**.

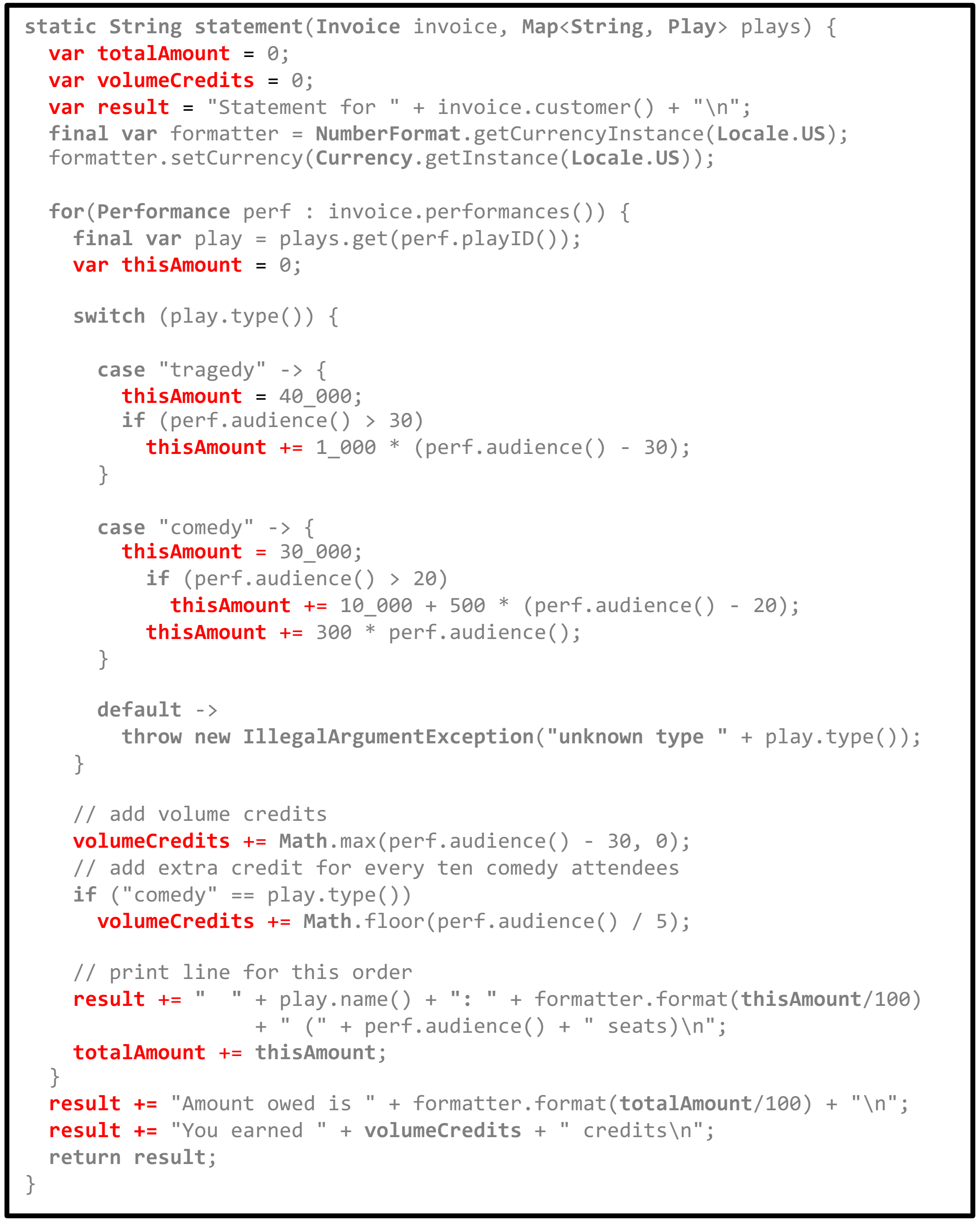
```
static String statement(Invoice invoice, Map<String, Play> plays) {
  var totalAmount = 0;
  var volumeCredits = 0;
  var result = "Statement for " + invoice.customer() + "\n";
  final var formatter = NumberFormat.getCurrencyInstance(Locale.US);
  formatter.setCurrency(Currency.getInstance(Locale.US));

  for(Performance perf : invoice.performances()) {
    final var play = plays.get(perf.playID());
    var thisAmount = 0;

    switch (play.type()) {

      case "tragedy" -> {
        thisAmount = 40_000;
        if (perf.audience() > 30)
          thisAmount +=1_000 * (perf.audience() - 30);
      }

      case "comedy" -> {
        thisAmount = 30_000;
          if (perf.audience() > 20)
            thisAmount += 10_000 + 500 * (perf.audience() - 20);
          thisAmount += 300 * perf.audience();
      }

      default ->
        throw new IllegalArgumentException("unknown type " + play.type());
    }

    // add volume credits
    volumeCredits += Math.max(perf.audience() - 30, 0);
    // add extra credit for every ten comedy attendees
    if ("comedy" == play.type())
      volumeCredits += Math.floor(perf.audience() / 5);

    // print line for this order
    result += "  " + play.name() + ": " + formatter.format(thisAmount/100)
                   + " (" + perf.audience() + " seats)\n";
    totalAmount += thisAmount;
  }
  result += "Amount owed is " + formatter.format(totalAmount/100) + "\n";
  result += "You earned " + volumeCredits + " credits\n";
  return result;
}
```

Here is the **Java** code again, together with the data structures we created earlier, and also a simple **regression test** consisting of a single **assertion**.

```
record Performance(String playID, int audience) { }

record Invoice(String customer, List<Performance> performances) { }

record Play(String name, String type) { }
```

```
static final List<Invoice> invoices =
  List.of(
    new Invoice(
      "BigCo",
      List.of(new Performance( "hamlet", 55),
              new Performance("as-like", 35),
              new Performance("othello", 40))));

static final Map<String,Play> plays = Map.of(
  "hamlet" , new Play("Hamlet", "tragedy"),
  "as-like", new Play("As You Like It", "comedy"),
  "othello", new Play("Othello", "tragedy"));
```

```
public static void main(String[] args) {
  if (!Statement.statement(invoices.get(0), plays).equals(
    """
    Statement for BigCo
      Hamlet: $650.00 (55 seats)
      As You Like It: $580.00 (35 seats)
      Othello: $500.00 (40 seats)
    Amount owed is $1,730.00
    You earned 47 credits
    """
  )) throw new AssertionError();
}
```



```
static String statement(Invoice invoice, Map<String, Play> plays) {
  var totalAmount = 0;
  var volumeCredits = 0;
  var result = "Statement for " + invoice.customer() + "\n";
  final var formatter = NumberFormat.getCurrencyInstance(Locale.US);
  formatter.setCurrency(Currency.getInstance(Locale.US));

  for(Performance perf : invoice.performances()) {
    final var play = plays.get(perf.playID());
    var thisAmount = 0;

    switch (play.type()) {

      case "tragedy" -> {
        thisAmount = 40_000;
        if (perf.audience() > 30)
          thisAmount += 1_000 * (perf.audience() - 30);
      }

      case "comedy" -> {
        thisAmount = 30_000;
          if (perf.audience() > 20)
            thisAmount += 10_000 + 500 * (perf.audience() - 20);
          thisAmount += 300 * perf.audience();
      }

      default ->
        throw new IllegalArgumentException("unknown type " + play.type());
    }

    // add volume credits
    volumeCredits += Math.max(perf.audience() - 30, 0);
    // add extra credit for every ten comedy attendees
    if ("comedy" == play.type())
      volumeCredits += Math.floor(perf.audience() / 5);

    // print line for this order
    result += "  " + play.name() + ": " + formatter.format(thisAmount/100)
                   + " (" + perf.audience() + " seats)\n";
    totalAmount += thisAmount;
  }
  result += "Amount owed is " + formatter.format(totalAmount/100) + "\n";
  result += "You earned " + volumeCredits + " credits\n";
  return result;
}
```

Yes, I hear you! Using **mutable variables** is best avoided when it is unecessary.

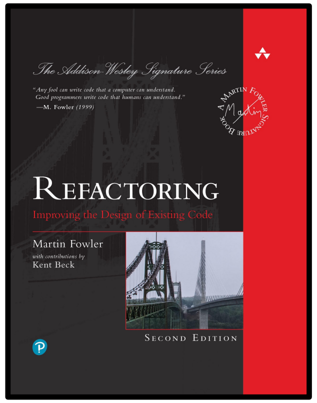
We are only using such variables in order to be faithful to **Martin Fowler**’s initial **Javascript** program.

Don’t worry: as we refactor the code, we’ll slowly but surely eliminate such **mutability**.

Martin Fowler

@martinfowler

Decomposing the `statement` Function


The Addison Wesley Signature Series
"Any fool can write code that a computer can understand. Good programmers write code that humans can understand."
—M. Fowler (1999)
A Martin Fowler Signature Book
REFACTORING
Improving the Design of Existing Code
Martin Fowler
with contributions by
Kent Beck
Second Edition

```java
static String statement(Invoice invoice, Map<String, Play> plays) {
  var totalAmount = 0;
  var volumeCredits = 0;
  var result = "Statement for " + invoice.customer() + "\n";
  final var formatter = NumberFormat.getCurrencyInstance(Locale.US);
  formatter.setCurrency(Currency.getInstance(Locale.US));

  for(Performance perf : invoice.performances()) {
    final var play = plays.get(perf.playID());
    var thisAmount = 0;

    switch (play.type()) {
      case "tragedy" -> {
        thisAmount = 40_000;
        if (perf.audience() > 30)
          thisAmount +=1_000 * (perf.audience() - 30);
      }
      case "comedy" -> {
        thisAmount = 30_000;
          if (perf.audience() > 20)
            thisAmount += 10_000 + 500 * (perf.audience() - 20);
          thisAmount += 300 * perf.audience();
      }
      default ->
        throw new IllegalArgumentException("unknown type " + play.type());
    }

    // add volume credits
    volumeCredits += Math.max(perf.audience() - 30, 0);
    // add extra credit for every ten comedy attendees
    if ("comedy" == play.type())
      volumeCredits += Math.floor(perf.audience() / 5);

    // print line for this order
    result += "  " + play.name() + ": " + formatter.format(thisAmount/100)
                      + " (" + perf.audience() + " seats)\n";
    totalAmount += thisAmount;
  }
  result += "Amount owed is " + formatter.format(totalAmount/100) + "\n";
  result += "You earned " + volumeCredits + " credits\n";
  return result;
}
```

When refactoring a long function like this, I mentally try to identify points that separate different parts of the overall behaviour.

The first chunk that leaps to my eye is the switch statement in the middle.

Martin Fowler

@martinfowler

```java
static String statement(Invoice invoice, Map<String, Play> plays) {
  var totalAmount = 0;
  var volumeCredits = 0;
  var result = "Statement for " + invoice.customer() + "\n";
  final var formatter = NumberFormat.getCurrencyInstance(Locale.US);
  formatter.setCurrency(Currency.getInstance(Locale.US));

  for(Performance perf : invoice.performances()) {
    final var play = plays.get(perf.playID());
    var thisAmount = 0;

    switch (play.type()) {
      case "tragedy" -> {
        thisAmount = 40_000;
        if (perf.audience() > 30)
          thisAmount +=1_000 * (perf.audience() - 30);
      }
      case "comedy" -> {
        thisAmount = 30_000;
          if (perf.audience() > 20)
            thisAmount += 10_000 + 500 * (perf.audience() - 20);
          thisAmount += 300 * perf.audience();
      }
      default ->
        throw new IllegalArgumentException("unknown type " + play.type());
    }

    // add volume credits
    volumeCredits += Math.max(perf.audience() - 30, 0);
    // add extra credit for every ten comedy attendees
    if ("comedy" == play.type())
      volumeCredits += Math.floor(perf.audience() / 5);

    // print line for this order
    result += "  " + play.name() + ": " + formatter.format(thisAmount/100)
                   + " (" + perf.audience() + " seats)\n";
    totalAmount += thisAmount;
  }
  result += "Amount owed is " + formatter.format(totalAmount/100) + "\n";
  result += "You earned " + volumeCredits + " credits\n";
  return result;
}
```

```java
static String statement(Invoice invoice, Map<String, Play> plays) {
  var totalAmount = 0;
  var volumeCredits = 0;
  var result = "Statement for " + invoice.customer() + "\n";
  final var formatter = NumberFormat.getCurrencyInstance(Locale.US);
  formatter.setCurrency(Currency.getInstance(Locale.US));

  for(Performance perf : invoice.performances()) {
    final var play = plays.get(perf.playID());
    final var thisAmount = amountFor.apply(perf, play);

    // add volume credits
    volumeCredits += Math.max(perf.audience() - 30, 0);
    // add extra credit for every ten comedy attendees
    if ("comedy" == play.type())
      volumeCredits += Math.floor(perf.audience() / 5);

    // print line for this order
    result += "  " + play.name() + ": " + formatter.format(thisAmount/100)
                   + " (" + perf.audience() + " seats)\n";
    totalAmount += thisAmount;
  }
  result += "Amount owed is " + formatter.format(totalAmount/100) + "\n";
  result += "You earned " + volumeCredits + " credits\n";
  return result;
}
```

- **Extract Function amountFor**
- In **amountFor** function:
  - rename **perf** arg to **aPerformance**
  - rename **thisAmount** arg to **result**

```java
BiFunction<Performance,Play,Integer> amountFor = (aPerformance, play) -> {
  var result = 0;
  switch (play.type()) {
    case "tragedy" -> {
      result = 40_000;
      if (aPerformance.audience() > 30)
        result += 1_000 * (aPerformance.audience() - 30);
    }
    case "comedy" -> {
      result = 30_000;
      if (aPerformance.audience() > 20)
        result += 10_000 + 500 * (aPerformance.audience() - 20);
      result += 300 * aPerformance.audience();
    }
    default ->
      throw new IllegalArgumentException("unknown type " + play.type());
  }
  return result;
}
```

```
static String statement(Invoice invoice, Map<String, Play> plays) {

  BiFunction<Performance,Play,Integer> amountFor = (aPerformance, play) -> {
    var result = 0;
    switch (play.type()) {
      case "tragedy" -> {
        result = 40_000;
        if (aPerformance.audience() > 30)
          result += 1_000 * (aPerformance.audience() - 30);
      }
      case "comedy" -> {
        result = 30_000;
        if (aPerformance.audience() > 20)
          result += 10_000 + 500 * (aPerformance.audience() - 20);
        result += 300 * aPerformance.audience();
      }
      default ->
        throw new IllegalArgumentException("unknown type " + play.type());
    }
    return result;
  };

  var totalAmount = 0;
  var volumeCredits = 0;
  var result = "Statement for " + invoice.customer() + "\n";
  final var formatter = NumberFormat.getCurrencyInstance(Locale.US);
  formatter.setCurrency(Currency.getInstance(Locale.US));

  for(Performance perf : invoice.performances()) {
    final var play = plays.get(perf.playID());
    final var thisAmount = amountFor.apply(perf, play);

    // add volume credits
    volumeCredits += Math.max(perf.audience() - 30, 0);
    // add extra credit for every ten comedy attendees
    if ("comedy" == play.type())
      volumeCredits += Math.floor(perf.audience() / 5);

    // print line for this order
    result += "  " + play.name() + ": " + formatter.format(thisAmount/100)
            + " (" + perf.audience() + " seats)\n";
    totalAmount += thisAmount;
  }
  result += "Amount owed is " + formatter.format(totalAmount/100) + "\n";
  result += "You earned " + volumeCredits + " credits\n";
  return result;
}
```

When Martin Fowler extracts a Javascript function from another, he nests the extracted child function inside the parent function from which it is extracted.

That makes sense, both to encapsulate (hide) the child function, which is just an implementation detail of the parent function, and to simplify the signature of a child function needing access to one or more parameters of the parent function, since nesting the child function means the parent's parameters are then directly accessible to it, rather than also having to be passed as parameters to it.

The problem is that in our case, the functions in question are Java methods, but Java does not directly support nested methods.

Because Martin Fowler's nesting of child functions is quite instrumental in his chosen refactoring approach, we are going to strike a compromise and define child functions as lambda functions, so that we are able to nest them inside their parent function.

The relatively small price that we'll pay for this compromise is that invoking lambda functions is more clunky than invoking methods (`f.apply(x)` rather than `f(x)`).

However, in the interest of clarity and brevity, I will at times show the statement function without also showing its child functions.

In the previous slide for example, although the amountFor function was extracted from statement, it is shown outside statement rather than nested inside it.

In the statement function on the left however, we do see amountFor nested inside statement.

@philip_schwarz

Martin Fowler
@martinfowler

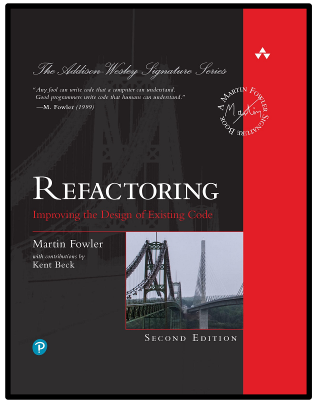
The next item to consider for renaming is the **play parameter**, but I have a **different fate** for that.

```
BiFunction<Performance,Play,Integer> amountFor = (aPerformance, play) -> {
  var result = 0;
  switch (play.type()) {
    case "tragedy" -> {
      result = 40_000;
      if (aPerformance.audience() > 30)
        result += 1_000 * (aPerformance.audience() - 30);
    }
    case "comedy" -> {
      result = 30_000;
      if (aPerformance.audience() > 20)
        result += 10_000 + 500 * (aPerformance.audience() - 20);
      result += 300 * aPerformance.audience();
    }
    default -> throw new IllegalArgumentException("unknown type " + play.type());
  }
  return result;
}
```

Martin Fowler

@martinfowler

- Decomposing the `statement` Function
  - Removing the `play` Variable


The Addison Wesley Signature Series
"Any fool can write code that a computer can understand. Good programmers write code that humans can understand."
—M. Fowler (1999)
A Martin Fowler Signature Book
REFACTORING
Improving the Design of Existing Code
Martin Fowler
with contributions by
Kent Beck
SECOND EDITION

Removing the play Variable

The next two slides perform a **Replace Temp with Query** **refactoring** on the **play variable**.

Such a **refactoring** is itself composed of the following **refactorings**:

- **Extract Function**
- **Inline Variable**

```
static String statement(Invoice invoice, Map<String, Play> plays) {
  var totalAmount = 0;
  var volumeCredits = 0;
  var result = "Statement for " + invoice.customer() + "\n";
  final var formatter = NumberFormat.getCurrencyInstance(Locale.US);
  formatter.setCurrency(Currency.getInstance(Locale.US));

  for(Performance perf : invoice.performances()) {
    final var play = plays.get(perf.playID());
    final var thisAmount = amountFor.apply(perf, play);

    // add volume credits
    volumeCredits += Math.max(perf.audience() - 30, 0);
    // add extra credit for every ten comedy attendees
    if ("comedy" == play.type())
      volumeCredits += Math.floor(perf.audience() / 5);

    // print line for this order
    result += "  " + play.name() + ": " + formatter.format(thisAmount/100)
                   + " (" + perf.audience() + " seats)\n";
    totalAmount += thisAmount;
  }
  result += "Amount owed is " + formatter.format(totalAmount/100) + "\n";
  result += "You earned " + volumeCredits + " credits\n";
  return result;
}
```

Removing the play Variable

```
static String statement(Invoice invoice, Map<String, Play> plays) {
  var totalAmount = 0;
  var volumeCredits = 0;
  var result = "Statement for " + invoice.customer() + "\n";
  final var formatter = NumberFormat.getCurrencyInstance(Locale.US);
  formatter.setCurrency(Currency.getInstance(Locale.US));

  for(Performance perf : invoice.performances()) {
    final var play = plays.get(perf.playID());
    final var thisAmount = amountFor.apply(perf, play);

    // add volume credits
    volumeCredits += Math.max(perf.audience() - 30, 0);
    // add extra credit for every ten comedy attendees
    if ("comedy" == play.type())
      volumeCredits += Math.floor(perf.audience() / 5);

    // print line for this order
    result += "  " + play.name() + ": " + formatter.format(thisAmount/100)
                   + " (" + perf.audience() + " seats)\n";
    totalAmount += thisAmount;
  }
  result += "Amount owed is " + formatter.format(totalAmount/100) + "\n";
  result += "You earned " + volumeCredits + " credits\n";
  return result;
}
```

- **Extract Function playFor**
- rename **playFor perf** parameter to **aPerformance**

```
Function<Performance,Play> playFor = aPerformance ->
  plays.get(aPerformance.playID());
```

```
static String statement(Invoice invoice, Map<String, Play> plays) {
  var totalAmount = 0;
  var volumeCredits = 0;
  var result = "Statement for " + invoice.customer() + "\n";
  final var formatter = NumberFormat.getCurrencyInstance(Locale.US);
  formatter.setCurrency(Currency.getInstance(Locale.US));

  for(Performance perf : invoice.performances()) {
    final var play = playFor.apply(perf);
    final var thisAmount = amountFor.apply(perf, play);

    // add volume credits
    volumeCredits += Math.max(perf.audience() - 30, 0);
    // add extra credit for every ten comedy attendees
    if ("comedy" == play.type())
      volumeCredits += Math.floor(perf.audience() / 5);

    // print line for this order
    result += "  " + play.name() + ": " + formatter.format(thisAmount/100)
                   + " (" + perf.audience() + " seats)\n";
    totalAmount += thisAmount;
  }
  result += "Amount owed is " + formatter.format(totalAmount/100) + "\n";
  result += "You earned " + volumeCredits + " credits\n";
  return result;
}
```

Removing the play Variable

```
static String statement(Invoice invoice, Map<String, Play> plays) {
  var totalAmount = 0;
  var volumeCredits = 0;
  var result = "Statement for " + invoice.customer() + "\n";
  final var formatter = NumberFormat.getCurrencyInstance(Locale.US);
  formatter.setCurrency(Currency.getInstance(Locale.US));

  for(Performance perf : invoice.performances()) {
    final var play = playFor.apply(perf);
    final var thisAmount = amountFor.apply(perf, play);

    // add volume credits
    volumeCredits += Math.max(perf.audience() - 30, 0);
    // add extra credit for every ten comedy attendees
    if ("comedy" == play.type())
      volumeCredits += Math.floor(perf.audience() / 5);

    // print line for this order
    result += "  " + play.name() + ": " + formatter.format(thisAmount/100)
               + " (" + perf.audience() + " seats)\n";
    totalAmount += thisAmount;
  }
  result += "Amount owed is " + formatter.format(totalAmount/100) + "\n";
  result += "You earned " + volumeCredits + " credits\n";
  return result;
}
```

Inline Variable **play** in **statement** function

```
static String statement(Invoice invoice, Map<String, Play> plays) {
  var totalAmount = 0;
  var volumeCredits = 0;
  var result = "Statement for " + invoice.customer() + "\n";
  final var formatter = NumberFormat.getCurrencyInstance(Locale.US);
  formatter.setCurrency(Currency.getInstance(Locale.US));

  for(Performance perf : invoice.performances()) {
    final var thisAmount = amountFor.apply(perf, playFor.apply(perf));

    // add volume credits
    volumeCredits += Math.max(perf.audience() - 30, 0);
    // add extra credit for every ten comedy attendees
    if ("comedy" == playFor.apply(perf).type())
      volumeCredits += Math.floor(perf.audience() / 5);

    // print line for this order
    result += "  " + playFor.apply(perf).name() + ": " + formatter.format(thisAmount/100)
               + " (" + perf.audience() + " seats)\n";
    totalAmount += thisAmount;
  }
  result += "Amount owed is " + formatter.format(totalAmount/100) + "\n";
  result += "You earned " + volumeCredits + " credits\n";
  return result;
}
```

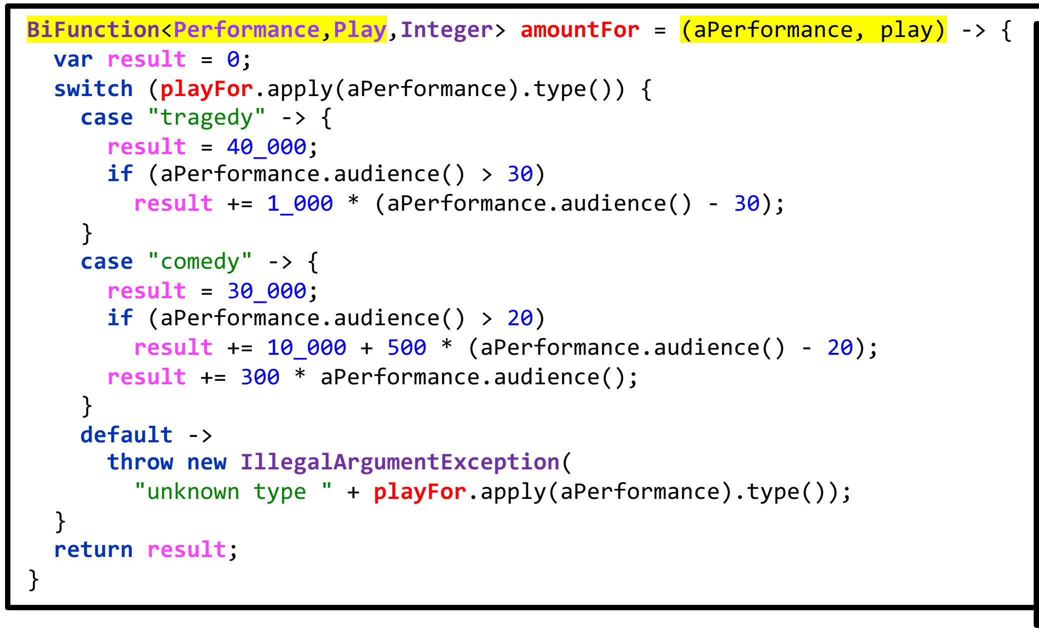
Removing the play Variable

in **amountFor** function: replace references to **play parameter** with invocations of **playFor function**

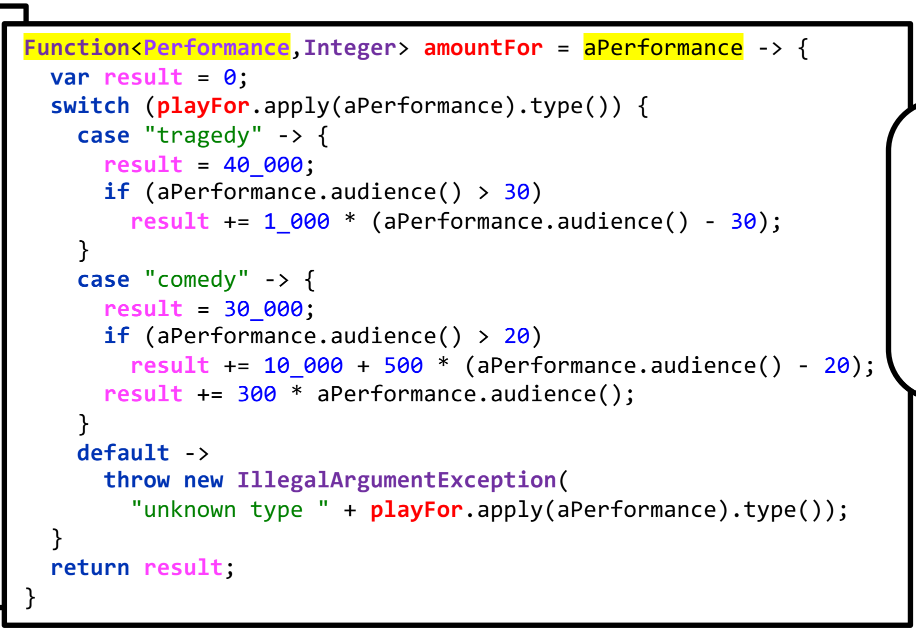
```
BiFunction<Performance,Play,Integer> amountFor = (aPerformance, play) -> {
  var result = 0;
  switch (play.type()) {
    case "tragedy" -> {
      result = 40_000;
      if (aPerformance.audience() > 30)
        result += 1_000 * (aPerformance.audience() - 30);
    }
    case "comedy" -> {
      result = 30_000;
      if (aPerformance.audience() > 20)
        result += 10_000 + 500 * (aPerformance.audience() - 20);
      result += 300 * aPerformance.audience();
    }
    default ->
      throw new IllegalArgumentException(
        "unknown type " + play.type());
  }
  return result;
}
```

```
BiFunction<Performance,Play,Integer> amountFor = (aPerformance, play) -> {
  var result = 0;
  switch (playFor.apply(aPerformance).type()) {
    case "tragedy" -> {
      result = 40_000;
      if (aPerformance.audience() > 30)
        result += 1_000 * (aPerformance.audience() - 30);
    }
    case "comedy" -> {
      result = 30_000;
      if (aPerformance.audience() > 20)
        result += 10_000 + 500 * (aPerformance.audience() - 20);
      result += 300 * aPerformance.audience();
    }
    default ->
      throw new IllegalArgumentException(
        "unknown type " + playFor.apply(aPerformance).type());
  }
  return result;
}
```

Removing the play Variable

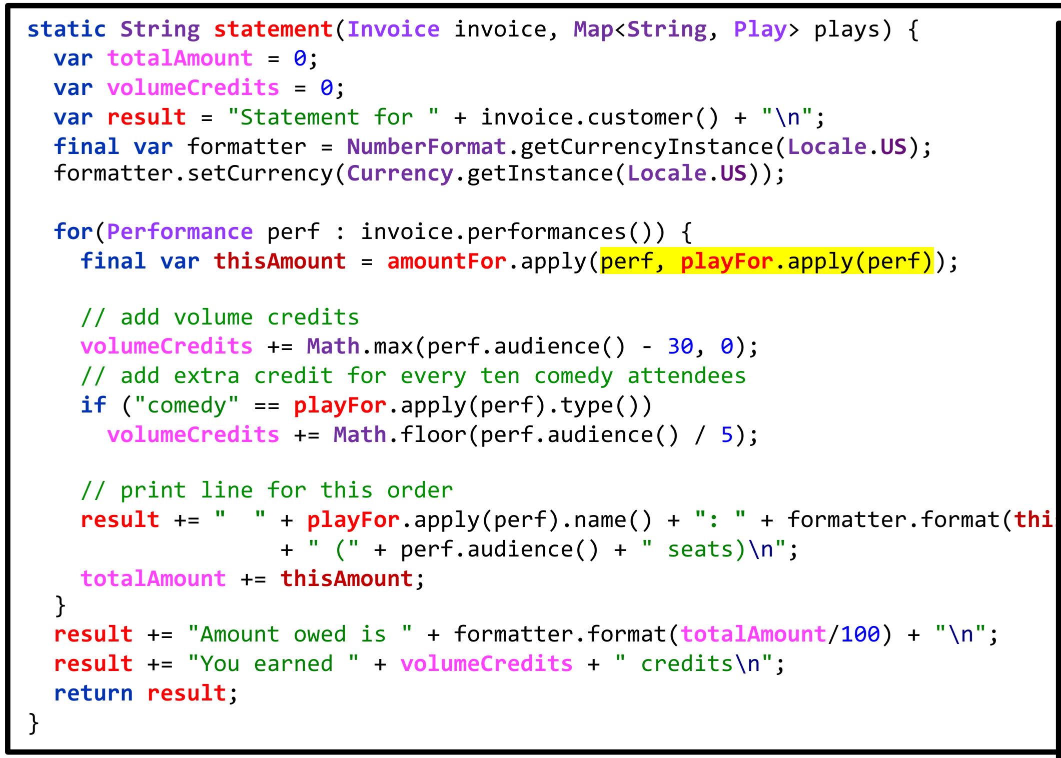


```
BiFunction<Performance,Play,Integer> amountFor = (aPerformance, play) -> {
  var result = 0;
  switch (playFor.apply(aPerformance).type()) {
    case "tragedy" -> {
      result = 40_000;
      if (aPerformance.audience() > 30)
        result += 1_000 * (aPerformance.audience() - 30);
    }
    case "comedy" -> {
      result = 30_000;
      if (aPerformance.audience() > 20)
        result += 10_000 + 500 * (aPerformance.audience() - 20);
      result += 300 * aPerformance.audience();
    }
    default ->
      throw new IllegalArgumentException(
        "unknown type " + playFor.apply(aPerformance).type());
  }
  return result;
}
```

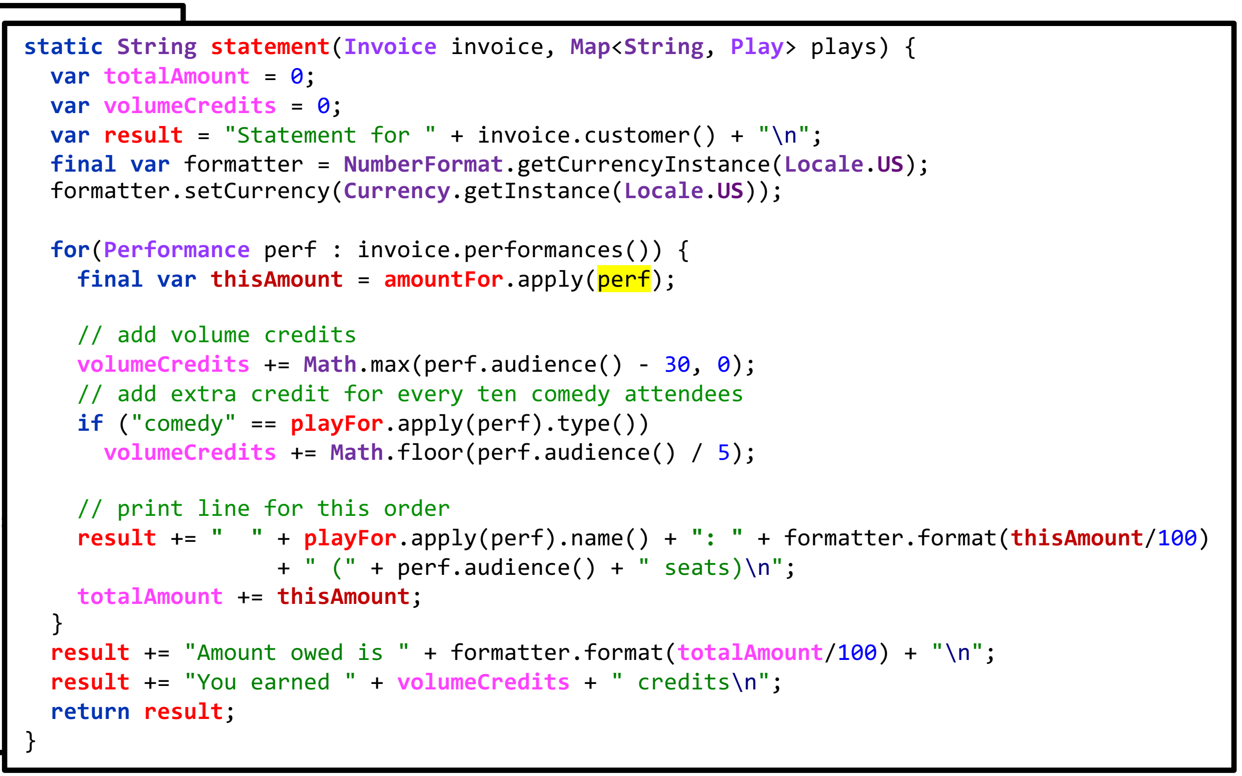


```
Function<Performance,Integer> amountFor = aPerformance -> {
  var result = 0;
  switch (playFor.apply(aPerformance).type()) {
    case "tragedy" -> {
      result = 40_000;
      if (aPerformance.audience() > 30)
        result += 1_000 * (aPerformance.audience() - 30);
    }
    case "comedy" -> {
      result = 30_000;
      if (aPerformance.audience() > 20)
        result += 10_000 + 500 * (aPerformance.audience() - 20);
      result += 300 * aPerformance.audience();
    }
    default ->
      throw new IllegalArgumentException(
        "unknown type " + playFor.apply(aPerformance).type());
  }
  return result;
}
```

Change Function Declaration of **amountFor** by removing **play parameter**



```
static String statement(Invoice invoice, Map<String, Play> plays) {
  var totalAmount = 0;
  var volumeCredits = 0;
  var result = "Statement for " + invoice.customer() + "\n";
  final var formatter = NumberFormat.getCurrencyInstance(Locale.US);
  formatter.setCurrency(Currency.getInstance(Locale.US));

  for(Performance perf : invoice.performances()) {
    final var thisAmount = amountFor.apply(perf, playFor.apply(perf));

    // add volume credits
    volumeCredits += Math.max(perf.audience() - 30, 0);
    // add extra credit for every ten comedy attendees
    if ("comedy" == playFor.apply(perf).type())
      volumeCredits += Math.floor(perf.audience() / 5);

    // print line for this order
    result += "  " + playFor.apply(perf).name() + ": " + formatter.format(thi
                  + " (" + perf.audience() + " seats)\n";
    totalAmount += thisAmount;
  }
  result += "Amount owed is " + formatter.format(totalAmount/100) + "\n";
  result += "You earned " + volumeCredits + " credits\n";
  return result;
}
```



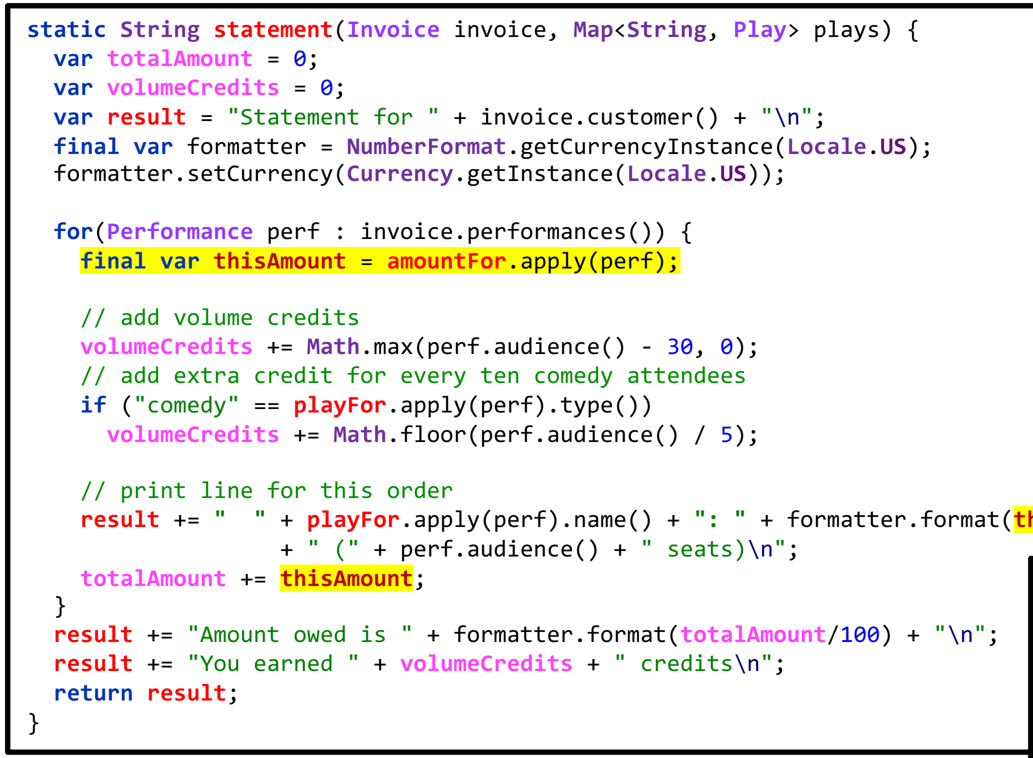
```
static String statement(Invoice invoice, Map<String, Play> plays) {
  var totalAmount = 0;
  var volumeCredits = 0;
  var result = "Statement for " + invoice.customer() + "\n";
  final var formatter = NumberFormat.getCurrencyInstance(Locale.US);
  formatter.setCurrency(Currency.getInstance(Locale.US));

  for(Performance perf : invoice.performances()) {
    final var thisAmount = amountFor.apply(perf);

    // add volume credits
    volumeCredits += Math.max(perf.audience() - 30, 0);
    // add extra credit for every ten comedy attendees
    if ("comedy" == playFor.apply(perf).type())
      volumeCredits += Math.floor(perf.audience() / 5);

    // print line for this order
    result += "  " + playFor.apply(perf).name() + ": " + formatter.format(thisAmount/100)
                  + " (" + perf.audience() + " seats)\n";
    totalAmount += thisAmount;
  }
  result += "Amount owed is " + formatter.format(totalAmount/100) + "\n";
  result += "You earned " + volumeCredits + " credits\n";
  return result;
}
```

Martin Fowler

@martinfowler

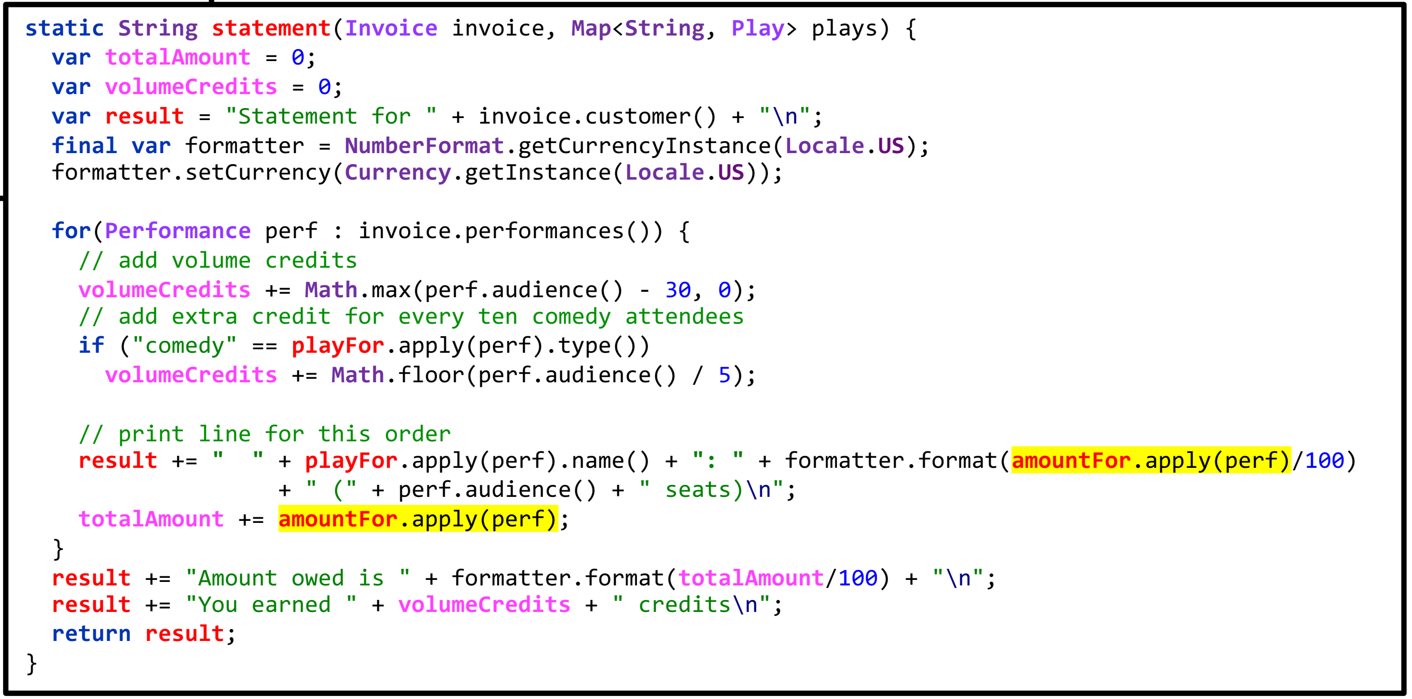
Now that I am done with the arguments to **amountFor**, I look back at where it’s called.

```
static String statement(Invoice invoice, Map<String, Play> plays) {
  var totalAmount = 0;
  var volumeCredits = 0;
  var result = "Statement for " + invoice.customer() + "\n";
  final var formatter = NumberFormat.getCurrencyInstance(Locale.US);
  formatter.setCurrency(Currency.getInstance(Locale.US));

  for(Performance perf : invoice.performances()) {
    final var thisAmount = amountFor.apply(perf);

    // add volume credits
    volumeCredits += Math.max(perf.audience() - 30, 0);
    // add extra credit for every ten comedy attendees
    if ("comedy" == playFor.apply(perf).type())
      volumeCredits += Math.floor(perf.audience() / 5);

    // print line for this order
    result += "  " + playFor.apply(perf).name() + ": " + formatter.format(thisAmount/100)
                   + " (" + perf.audience() + " seats)\n";
    totalAmount += thisAmount;
  }
  result += "Amount owed is " + formatter.format(totalAmount/100) + "\n";
  result += "You earned " + volumeCredits + " credits\n";
  return result;
}
```



```
static String statement(Invoice invoice, Map<String, Play> plays) {
  var totalAmount = 0;
  var volumeCredits = 0;
  var result = "Statement for " + invoice.customer() + "\n";
  final var formatter = NumberFormat.getCurrencyInstance(Locale.US);
  formatter.setCurrency(Currency.getInstance(Locale.US));

  for(Performance perf : invoice.performances()) {
    final var thisAmount = amountFor.apply(perf);

    // add volume credits
    volumeCredits += Math.max(perf.audience() - 30, 0);
    // add extra credit for every ten comedy attendees
    if ("comedy" == playFor.apply(perf).type())
      volumeCredits += Math.floor(perf.audience() / 5);

    // print line for this order
    result += "  " + playFor.apply(perf).name() + ": " + formatter.format(thisAmount/100)
              + " (" + perf.audience() + " seats)\n";
    totalAmount += thisAmount;
  }
  result += "Amount owed is " + formatter.format(totalAmount/100) + "\n";
  result += "You earned " + volumeCredits + " credits\n";
  return result;
}
```

Inline Variable thisAmount in statement function

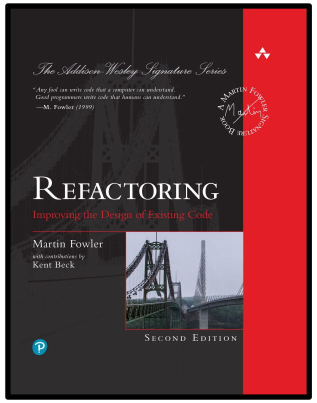


```
static String statement(Invoice invoice, Map<String, Play> plays) {
  var totalAmount = 0;
  var volumeCredits = 0;
  var result = "Statement for " + invoice.customer() + "\n";
  final var formatter = NumberFormat.getCurrencyInstance(Locale.US);
  formatter.setCurrency(Currency.getInstance(Locale.US));

  for(Performance perf : invoice.performances()) {
    // add volume credits
    volumeCredits += Math.max(perf.audience() - 30, 0);
    // add extra credit for every ten comedy attendees
    if ("comedy" == playFor.apply(perf).type())
      volumeCredits += Math.floor(perf.audience() / 5);

    // print line for this order
    result += "  " + playFor.apply(perf).name() + ": " + formatter.format(amountFor.apply(perf)/100)
              + " (" + perf.audience() + " seats)\n";
    totalAmount += amountFor.apply(perf);
  }
  result += "Amount owed is " + formatter.format(totalAmount/100) + "\n";
  result += "You earned " + volumeCredits + " credits\n";
  return result;
}
```

Martin Fowler

@martinfowler

- Decomposing the `statement` Function
  - Removing the play Variable
  - Extracting Volume Credits



Extracting Volume Credits

Martin Fowler

@martinfowler

Now I get the **benefit** from removing the **play variable** as it makes it easier to extract the **volume credits** calculation by removing one of the locally scoped variables. I still have to deal with the other two.

```java
static String statement(Invoice invoice, Map<String, Play> plays) {
  var totalAmount = 0;
  var volumeCredits = 0;
  var result = "Statement for " + invoice.customer() + "\n";
  final var formatter = NumberFormat.getCurrencyInstance(Locale.US);
  formatter.setCurrency(Currency.getInstance(Locale.US));

  for(Performance perf : invoice.performances()) {
    // add volume credits
    volumeCredits += Math.max(perf.audience() - 30, 0);
    // add extra credit for every ten comedy attendees
    if ("comedy" == playFor.apply(perf).type())
      volumeCredits += Math.floor(perf.audience() / 5);

    // print line for this order
    result += "  " + playFor.apply(perf).name() + ": " + formatter.format(amountFor.apply(perf)/100)
                   + " (" + perf.audience() + " seats)\n";
    totalAmount += amountFor.apply(perf);
  }
  result += "Amount owed is " + formatter.format(totalAmount/100) + "\n";
  result += "You earned " + volumeCredits + " credits\n";
  return result;
}
```

Extracting Volume Credits

```java
static String statement(Invoice invoice, Map<String, Play> plays) {
  var totalAmount = 0;
  var volumeCredits = 0;
  var result = "Statement for " + invoice.customer() + "\n";
  final var formatter = NumberFormat.getCurrencyInstance(Locale.US);
  formatter.setCurrency(Currency.getInstance(Locale.US));

  for(Performance perf : invoice.performances()) {
    // add volume credits
    volumeCredits += Math.max(perf.audience() - 30, 0);
    // add extra credit for every ten comedy attendees
    if ("comedy" == playFor.apply(perf).type())
      volumeCredits += Math.floor(perf.audience() / 5);

    // print line for this order
    result += "  " + playFor.apply(perf).name() + ": " + formatter.format(amountFor.apply(perf)/100)
              + " (" + perf.audience() + " seats)\n";
    totalAmount += amountFor.apply(perf);
  }
  result += "Amount owed is " + formatter.format(totalAmount/100) + "\n"
  result += "You earned " + volumeCredits + " credits\n";
  return result;
}
```

- **Extract Function volumeCreditsFor**
- In **volumeCreditsFor** function:
  - rename **perf** arg to **aPerformance**
  - rename **volumeCredits** arg to **result**

```java
static String statement(Invoice invoice, Map<String, Play> plays) {
  var totalAmount = 0;
  var volumeCredits = 0;
  var result = "Statement for " + invoice.customer() + "\n";
  final var formatter = NumberFormat.getCurrencyInstance(Locale.US);
  formatter.setCurrency(Currency.getInstance(Locale.US));

  for(Performance perf : invoice.performances()) {
    volumeCredits += volumeCreditsFor.apply(perf);

    // print line for this order
    result += "  " + playFor.apply(perf).name() + ": " + formatter.format(amountFor.apply(perf)/100)
              + " (" + perf.audience() + " seats)\n";
    totalAmount += amountFor.apply(perf);
  }
  result += "Amount owed is " + formatter.format(totalAmount/100) + "\n";
  result += "You earned " + volumeCredits + " credits\n";
  return result;
}
```

```java
Function<Performance,Integer> volumeCreditsFor = aPerformance -> {
  var result = 0;
  result += Math.max(aPerformance.audience() - 30, 0);
  if ("comedy" == playFor.apply(aPerformance).type())
    result += Math.floor(aPerformance.audience() / 5);
  return result;
};
```

Martin Fowler
@martinfowler

As I suggested before, **temporary variables** can be a problem. They are only useful within their own routine, and therefore encourage **long**, **complex routines**.

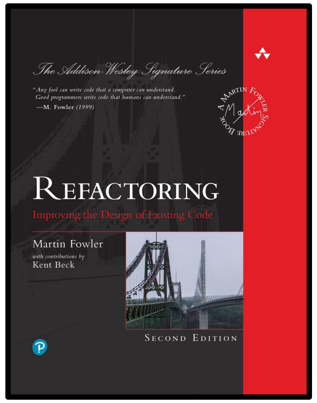
My next move, then, is to replace some of them. The easiest one is **formatter**.

```
static String statement(Invoice invoice, Map<String, Play> plays) {
  var totalAmount = 0;
  var volumeCredits = 0;
  var result = "Statement for " + invoice.customer() + "\n";
  final var formatter = NumberFormat.getCurrencyInstance(Locale.US);
  formatter.setCurrency(Currency.getInstance(Locale.US));

  for(Performance perf : invoice.performances()) {
    volumeCredits += volumeCreditsFor.apply(perf);

    // print line for this order
    result += "  " + playFor.apply(perf).name() + ": " + formatter.format(amountFor.apply(perf)/100)
                   + " (" + perf.audience() + " seats)\n";
    totalAmount += amountFor.apply(perf);
  }
  result += "Amount owed is " + formatter.format(totalAmount/100) + "\n";
  result += "You earned " + volumeCredits + " credits\n";
  return result;
}
```

Martin Fowler

@martinfowler

- Decomposing the `statement` Function
  - Removing the play Variable
  - Extracting Volume Credits
  - Removing the `formatter` Variable


The Addison-Wesley Signature Series
"Any fool can write code that a computer can understand. Good programmers write code that humans can understand."
—M. Fowler (1999)
A Martin Fowler Signature Book
Refactoring
Improving the Design of Existing Code
Martin Fowler
with contributions by
Kent Beck
Second Edition

Removing the `formatter` Variable

```java
static String statement(Invoice invoice, Map<String, Play> plays) {
  var totalAmount = 0;
  var volumeCredits = 0;
  var result = "Statement for " + invoice.customer() + "\n";
  final var formatter = NumberFormat.getCurrencyInstance(Locale.US);
  formatter.setCurrency(Currency.getInstance(Locale.US));

  for(Performance perf : invoice.performances()) {
    volumeCredits += volumeCreditsFor.apply(perf);

    // print line for this order
    result += "  " + playFor.apply(perf).name() + ": " + formatter.format(amountFor.apply(perf)/100)
              + " (" + perf.audience() + " seats)\n";
    totalAmount += amountFor.apply(perf);
  }
  result += "Amount owed is " + formatter.format(totalAmount/100) + "\n";
  result += "You earned " + volumeCredits + " credits\n";
  return result;
}
```

- Extract Function **format**
- Replace references to **formatter.format** with invocations of **format**
- Change Function Declaration of **format** by renaming function to **usd**

```java
static String statement(Invoice invoice, Map<String, Play> plays) {
  var totalAmount = 0;
  var volumeCredits = 0;
  var result = "Statement for " + invoice.customer() + "\n";

  for(Performance perf : invoice.performances()) {
    volumeCredits += volumeCreditsFor.apply(perf);

    // print line for this order
    result += "  " + playFor.apply(perf).name() + ": " + usd.apply(amountFor.apply(perf)/100)
              + " (" + perf.audience() + " seats)\n";
    totalAmount += amountFor.apply(perf);
  }
  result += "Amount owed is " + usd.apply(totalAmount/100) + "\n";
  result += "You earned " + volumeCredits + " credits\n";
  return result;
}
```

```java
Function<Integer,String> usd = aNumber -> {
  final var formatter = NumberFormat.getCurrencyInstance(Locale.US);
  formatter.setCurrency(Currency.getInstance(Locale.US));
  return formatter.format(aNumber);
};
```

Martin Fowler

@martinfowler

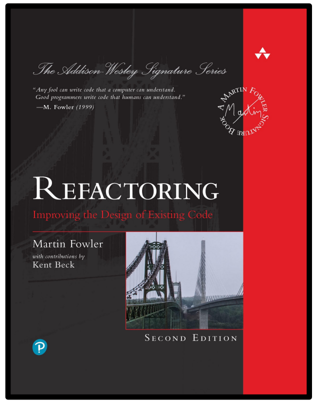
My next terget variable is **volumeCredits**. This is a trickier case, as it’s built up during the iterations of the loop.

```
static String statement(Invoice invoice, Map<String, Play> plays) {
  var totalAmount = 0;
  var volumeCredits = 0;
  var result = "Statement for " + invoice.customer() + "\n";

  for(Performance perf : invoice.performances()) {
    volumeCredits += volumeCreditsFor.apply(perf);

    // print line for this order
    result += "  " + playFor.apply(perf).name() + ": " + usd.apply(amountFor.apply(perf)/100)
                   + " (" + perf.audience() + " seats)\n";
    totalAmount += amountFor.apply(perf);
  }
  result += "Amount owed is " + usd.apply(totalAmount/100) + "\n";
  result += "You earned " + volumeCredits + " credits\n";
  return result;
}
```

Martin Fowler

@martinfowler

- Decomposing the `statement` Function
  - Removing the play Variable
  - Extracting Volume Credits
  - Removing the `formatter` Variable
  - Removing Total Volume Credits


The Addison-Wesley Signature Series
"Any fool can write code that a computer can understand. Good programmers write code that humans can understand."
—M. Fowler (1999)
A Martin Fowler Signature Book
Refactoring
Improving the Design of Existing Code
Martin Fowler
with contributions by
Kent Beck
Second Edition

Removing Total Volume Credits

```
static String statement(Invoice invoice, Map<String, Play> plays) {
  var totalAmount = 0;
  var volumeCredits = 0;
  var result = "Statement for " + invoice.customer() + "\n";

  for(Performance perf : invoice.performances()) {
    volumeCredits += volumeCreditsFor.apply(perf);

    // print line for this order
    result += "  " + playFor.apply(perf).name() + ": " + usd.apply(amountFor.apply(perf)/100)
               + " (" + perf.audience() + " seats)\n";
    totalAmount += amountFor.apply(perf);
  }
  result += "Amount owed is " + usd.apply(totalAmount/100) + "\n";
  result += "You earned " + volumeCredits + " credits\n";
  return result;
}
```

- Apply **Split Loop** to the loop on invoice.performances
- Apply **Slide Statements** to the statement initialising variable volumeCredits

```
static String statement(Invoice invoice, Map<String, Play> plays) {
  var totalAmount = 0;
  var result = "Statement for " + invoice.customer() + "\n";

  for(Performance perf : invoice.performances()) {
    // print line for this order
    result += "  " + playFor.apply(perf).name() + ": " + usd.apply(amountFor.apply(perf)/100)
               + " (" + perf.audience() + " seats)\n";
    totalAmount += amountFor.apply(perf);
  }

  var volumeCredits = 0;
  for(Performance perf : invoice.performances()) {
    volumeCredits += volumeCreditsFor.apply(perf);
  }

  result += "Amount owed is " + usd.apply(totalAmount/100) + "\n";
  result += "You earned " + volumeCredits + " credits\n";
  return result;
}
```

Removing Total Volume Credits

The next two slides perform a **Replace Temp with Query refactoring** on the volumeCredits variable.

As we saw earlier on, such a **refactoring** is itself composed of the following **refactorings**:

- **Extract Function**
- **Inline Variable**

```
static String statement(Invoice invoice, Map<String, Play> plays) {
  var totalAmount = 0;
  var result = "Statement for " + invoice.customer() + "\n";

  for(Performance perf : invoice.performances()) {
    // print line for this order
    result += "  " + playFor.apply(perf).name() + ": " + usd.apply(amountFor.apply(perf)/100)
                   + " (" + perf.audience() + " seats)\n";
    totalAmount += amountFor.apply(perf);
  }

  var volumeCredits = 0;
  for(Performance perf : invoice.performances()) {
    volumeCredits += volumeCreditsFor.apply(perf);
  }

  result += "Amount owed is " + usd.apply(totalAmount/100) + "\n";
  result += "You earned " + volumeCredits + " credits\n";
  return result;
}
```

Removing Total Volume Credits

```java
static String statement(Invoice invoice, Map<String, Play> plays) {
  var totalAmount = 0;
  var result = "Statement for " + invoice.customer() + "\n";

  for(Performance perf : invoice.performances()) {
    // print line for this order
    result += "  " + playFor.apply(perf).name() + ": " + usd.apply(amountFor.apply(perf)/100)
                   + " (" + perf.audience() + " seats)\n";
    totalAmount += amountFor.apply(perf);
  }

  var volumeCredits = 0;
  for(Performance perf : invoice.performances()) {
    volumeCredits += volumeCreditsFor.apply(perf);
  }

  result += "Amount owed is " + usd.apply(totalAmount/100) + "\n";
  result += "You earned " + volumeCredits + " credits\n";
  return result;
}
```

- **Extract Function totalVolumeCredits**
- **Inline Variable volumeCredits**

```java
static String statement(Invoice invoice, Map<String, Play> plays) {
  var totalAmount = 0;
  var result = "Statement for " + invoice.customer() + "\n";

  for(Performance perf : invoice.performances()) {
    // print line for this order
    result += "  " + playFor.apply(perf).name() + ": " + usd.apply(amountFor.apply(perf)/100)
                   + " (" + perf.audience() + " seats)\n";
    totalAmount += amountFor.apply(perf);
  }

  result += "Amount owed is " + usd.apply(totalAmount/100) + "\n";
  result += "You earned " + totalVolumeCredits.get() + " credits\n";
  return result;
}
```

```java
Supplier<Integer> totalVolumeCredits = () -> {
  var volumeCredits = 0;
  for (Performance perf : invoice.performances()) {
    volumeCredits += volumeCreditsFor.apply(perf);
  }
  return volumeCredits;
};
```

Martin Fowler

@martinfowler

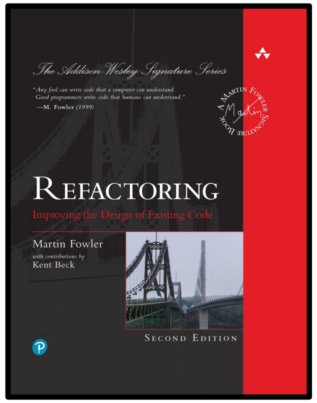
I then repeat that sequence to remove **totalAmount**.

```java
static String statement(Invoice invoice, Map<String, Play> plays) {
  var totalAmount = 0;
  var result = "Statement for " + invoice.customer() + "\n";

  for(Performance perf : invoice.performances()) {
    // print line for this order
    result += "  " + playFor.apply(perf).name() + ": " + usd.apply(amountFor.apply(perf)/100)
                   + " (" + perf.audience() + " seats)\n";
    totalAmount += amountFor.apply(perf);
  }

  result += "Amount owed is " + usd.apply(totalAmount/100) + "\n";
  result += "You earned " + totalVolumeCredits.get() + " credits\n";
  return result;
}
```

Martin Fowler

@martinfowler

- Decomposing the `statement` Function
  - Removing the play Variable
  - Extracting Volume Credits
  - Removing the `formatter` Variable
  - Removing Total Volume Credits
  - Removing Total Amount


The Addison-Wesley Signature Series
"Any fool can write code that a computer can understand. Good programmers write code that humans can understand."
—M. Fowler (1999)
A Martin Fowler Signature Book
Martin
REFACTORING
Improving the Design of Existing Code
Martin Fowler
with contributions by
Kent Beck
SECOND EDITION

Removing Total Amount

```
static String statement(Invoice invoice, Map<String, Play> plays) {
  var totalAmount = 0;
  var result = "Statement for " + invoice.customer() + "\n";

  for(Performance perf : invoice.performances()) {
    // print line for this order
    result += "  " + playFor.apply(perf).name() + ": " + usd.apply(amountFor.apply(perf)/100)
                    + " (" + perf.audience() + " seats)\n";
    totalAmount += amountFor.apply(perf);
  }

  result += "Amount owed is " + usd.apply(totalAmount/100) + "\n";
  result += "You earned " + totalVolumeCredits.get() + " credits\n";
  return result;
}
```

- Apply **Split Loop** to the loop on invoice.performances
- Apply **Slide Statements** to the statement initialising variable **totalAmount**

```
static String statement(Invoice invoice, Map<String, Play> plays) {
  var result = "Statement for " + invoice.customer() + "\n";
  for(Performance perf : invoice.performances()) {
    result += "  " + playFor.apply(perf).name() + ": " + usd.apply(amountFor.apply(perf)/100)
                   + " (" + perf.audience() + " seats)\n";
  }

  var totalAmount = 0;
  for(Performance perf : invoice.performances()) {
    totalAmount += amountFor.apply(perf);
  }

  result += "Amount owed is " + usd.apply(totalAmount/100) + "\n";
  result += "You earned " + totalVolumeCredits.get() + " credits\n";
  return result;
}
```

Removing Total Amount

```
static String statement(Invoice invoice, Map<String, Play> plays) {
  var result = "Statement for " + invoice.customer() + "\n";
  for(Performance perf : invoice.performances()) {
    result += "  " + playFor.apply(perf).name() + ": " + usd.apply(amountFor.apply(perf)/100)
                   + " (" + perf.audience() + " seats)\n";
  }

  var totalAmount = 0;
  for(Performance perf : invoice.performances()) {
    totalAmount += amountFor.apply(perf);
  }

  result += "Amount owed is " + usd.apply(totalAmount/100) + "\n";
  result += "You earned " + totalVolumeCredits.get() + " credits\n";
  return result;
}
```

- **Extract Function appleSauce**
- **Inline Variable totalAmount**

```
static String statement(Invoice invoice, Map<String, Play> plays) {
  var result = "Statement for " + invoice.customer() + "\n";
  for(Performance perf : invoice.performances()) {
    result += "  " + playFor.apply(perf).name() + ": " + usd.apply(amountFor.apply(perf)/100)
                   + " (" + perf.audience() + " seats)\n";
  }

  result += "Amount owed is " + usd.apply(appleSauce.get()/100) + "\n";
  result += "You earned " + totalVolumeCredits.get() + " credits\n";
  return result;
}
```

```
Supplier<Integer> appleSauce = () -> {
  var totalAmount = 0;
  for (Performance perf : invoice.performances())
    totalAmount += amountFor.apply(perf);
  return totalAmount;
};
```

Removing Total Amount

```java
static String statement(Invoice invoice, Map<String, Play> plays) {
  var result = "Statement for " + invoice.customer() + "\n";
  for(Performance perf : invoice.performances()) {
    result += "  " + playFor.apply(perf).name() + ": " + usd.apply(amountFor.apply(perf)/100)
                   + " (" + perf.audience() + " seats)\n";
  }

  result += "Amount owed is " + usd.apply(appleSauce.get()/100) + "\n";
  result += "You earned " + totalVolumeCredits.get() + " credits\n";
  return result;
}
```

- **Change Function Declaration** of **appleSauce** by renaming function to **totalAmount**
- **Rename Variables** volumeCredits and totalAmount to result

```java
Supplier<Integer> totalVolumeCredits = () -> {
  var volumeCredits = 0;
  for (Performance perf : invoice.performances())
    volumeCredits += volumeCreditsFor.apply(perf);
  return volumeCredits;
};
```

```java
Supplier<Integer> appleSauce = () -> {
  var totalAmount = 0;
  for (Performance perf : invoice.performances())
    totalAmount += amountFor.apply(perf);
  return totalAmount;
};
```

```java
static String statement(Invoice invoice, Map<String, Play> plays) {
  var result = "Statement for " + invoice.customer() + "\n";
  for(Performance perf : invoice.performances()) {
    result += "  " + playFor.apply(perf).name() + ": " + usd.apply(amountFor.apply(perf)/100)
                   + " (" + perf.audience() + " seats)\n";
  }

  result += "Amount owed is " + usd.apply(totalAmount.get()/100) + "\n";
  result += "You earned " + totalVolumeCredits.get() + " credits\n";
  return result;
}
```

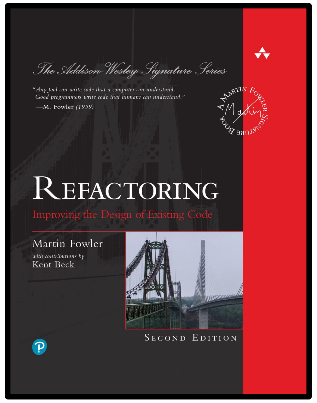
```java
Supplier<Integer> totalVolumeCredits = () -> {
  var result = 0;
  for (Performance perf : invoice.performances())
    result += volumeCreditsFor.apply(perf);
  return result;
};
```

```java
Supplier<Integer> totalAmount = () -> {
  var result = 0;
  for (Performance perf : invoice.performances())
    result += amountFor.apply(perf);
  return result;
};
```

Martin Fowler

@martinfowler

- Decomposing the `statement` Function
  - Removing the play Variable
  - Extracting Volume Credits
  - Removing the `formatter` Variable
  - Removing Total Volume Credits
  - Removing Total Amount
  - Status: Lots of Nested Functions


The Addison Wesley Signature Series
Refactoring
Improving the Design of Existing Code
Martin Fowler
with contributions by
Kent Beck
Second Edition

Status: Lots of Nested Functions

Now is a good time to pause and take a look at the overall state of the code.

The structure of the code is much better now.

The top-level statement function is now just six lines of code, and all it does is laying out the printing of the statement.

All the calculation logic has been moved out to a handful of supporting functions.

This makes it easier to understand each individual calculation as well as the overall flow of the report.

Martin Fowler
@martinfowler

```
static String statement(Invoice invoice, Map<String, Play> plays) {
  Function<Performance,Play> playFor = aPerformance -> plays.get(aPerformance.playID());
  Function<Performance,Integer> amountFor = aPerformance -> {
    var result = 0;
    switch (playFor.apply(aPerformance).type()) {
      case "tragedy" -> {
        result = 40_000;
        if (aPerformance.audience() > 30) result += 1_000 * (aPerformance.audience() - 30);
      }
      case "comedy" -> {
        result = 30_000;
        if (aPerformance.audience() > 20) result += 10_000 + 500 * (aPerformance.audience() - 20);
        result += 300 * aPerformance.audience();
      }
      default ->
        throw new IllegalArgumentException("unknown type " + playFor.apply(aPerformance).type());
    }
    return result;
  };
  Function<Performance,Integer> volumeCreditsFor = aPerformance -> {
    var result = 0;
    result += Math.max(aPerformance.audience() - 30, 0);
    if ("comedy" == playFor.apply(aPerformance).type()) result += Math.floor(aPerformance.audience() / 5);
    return result;
  };
  Function<Integer,String> usd = aNumber -> {
    final var formatter = NumberFormat.getCurrencyInstance(Locale.US);
    formatter.setCurrency(Currency.getInstance(Locale.US));
    return formatter.format(aNumber);
  };
  Supplier<Integer> totalVolumeCredits = () -> {
    var result = 0;
    for (Performance perf : invoice.performances())
      result += volumeCreditsFor.apply(perf);
    return result;
  };
  Supplier<Integer> totalAmount = () -> {
    var result = 0;
    for (Performance perf : invoice.performances())
      result += amountFor.apply(perf);
    return result;
  };
  var result = "Statement for " + invoice.customer() + "\n";
  for(Performance perf : invoice.performances())
    result += "  " + playFor.apply(perf).name() + ": " + usd.apply(amountFor.apply(perf)/100)
                   + " (" + perf.audience() + " seats)\n";
  result += "Amount owed is " + usd.apply(totalAmount.get()/100) + "\n";
  result += "You earned " + totalVolumeCredits.get() + " credits\n";
  return result;
}
```

Original Program

```
static String statement(Invoice invoice, Map<String, Play> plays) {
  var totalAmount = 0;
  var volumeCredits = 0;
  var result = "Statement for " + invoice.customer() + "\n";
  final var formatter = NumberFormat.getCurrencyInstance(Locale.US);
  formatter.setCurrency(Currency.getInstance(Locale.US));

  for(Performance perf : invoice.performances()) {
    final var play = plays.get(perf.playID());
    var thisAmount = 0;
    switch (play.type()) {
      case "tragedy" -> {
        thisAmount = 40_000;
        if (perf.audience() > 30) thisAmount +=1_000 * (perf.audience() - 30);
      }
      case "comedy" -> {
        thisAmount = 30_000;
          if (perf.audience() > 20) thisAmount += 10_000 + 500 * (perf.audience() - 20);
          thisAmount += 300 * perf.audience();
      }
      default -> throw new IllegalArgumentException("unknown type " + play.type());
    }

    // add volume credits
    volumeCredits += Math.max(perf.audience() - 30, 0);
    // add extra credit for every ten comedy attendees
    if ("comedy" == play.type()) volumeCredits += Math.floor(perf.audience() / 5);

    // print line for this order
    result += "  " + play.name() + ": " + formatter.format(thisAmount/100)
               + " (" + perf.audience() + " seats)\n";
    totalAmount += thisAmount;
  }
  result += "Amount owed is " + formatter.format(totalAmount/100) + "\n";
  result += "You earned " + volumeCredits + " credits\n";
  return result;
}
```

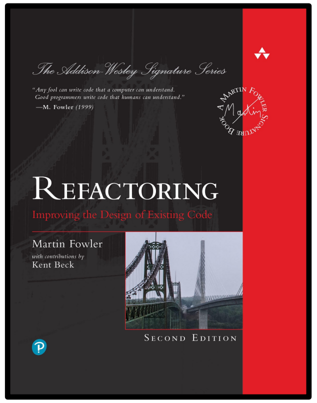
Refactored Program

```
static String statement(Invoice invoice, Map<String, Play> plays) {
  Function<Performance,Play> playFor = aPerformance -> plays.get(aPerformance.playID());
  Function<Performance,Integer> amountFor = aPerformance -> {
    var result = 0;
    switch (playFor.apply(aPerformance).type()) {
      case "tragedy" -> {
        result = 40_000;
        if (aPerformance.audience() > 30) result += 1_000 * (aPerformance.audience() - 30);
      }
      case "comedy" -> {
        result = 30_000;
        if (aPerformance.audience() > 20) result += 10_000 + 500 * (aPerformance.audience() - 20);
        result += 300 * aPerformance.audience();
      }
      default -> throw new IllegalArgumentException("unknown type " + playFor.apply(aPerformance).type());
    }
    return result;
  };
  Function<Performance,Integer> volumeCreditsFor = aPerformance -> {
    var result = 0;
    result += Math.max(aPerformance.audience() - 30, 0);
    if ("comedy" == playFor.apply(aPerformance).type()) result += Math.floor(aPerformance.audience() / 5);
    return result;
  };
  Function<Integer,String> usd = aNumber -> {
    final var formatter = NumberFormat.getCurrencyInstance(Locale.US);
    formatter.setCurrency(Currency.getInstance(Locale.US));
    return formatter.format(aNumber);
  };
  Supplier<Integer> totalVolumeCredits = () -> {
    var result = 0;
    for (Performance perf : invoice.performances())
      result += volumeCreditsFor.apply(perf);
    return result;
  };
  Supplier<Integer> totalAmount = () -> {
    var result = 0;
    for (Performance perf : invoice.performances())
      result += amountFor.apply(perf);
    return result;
  };
  var result = "Statement for " + invoice.customer() + "\n";
  for(Performance perf : invoice.performances())
    result += "  " + playFor.apply(perf).name() + ": " + usd.apply(amountFor.apply(perf)/100)
               + " (" + perf.audience() + " seats)\n";
  result += "Amount owed is " + usd.apply(totalAmount.get()/100) + "\n";
  result += "You earned " + totalVolumeCredits.get() + " credits\n";
  return result;
}
```

Martin Fowler

@martinfowler

- Decomposing the `statement` Function
  - Removing the play Variable
  - Extracting Volume Credits
  - Removing the `formatter` Variable
  - Removing Total Volume Credits
  - Removing Total Amount
  - Status: Lots of Nested Functions
  - Splitting the Phases of Calculation and Formatting


The Addison-Wesley Signature Series
Refactoring
Improving the Design of Existing Code
Martin Fowler
with contributions by
Kent Beck
Second Edition

Splitting the Phases of Calculation and Formatting

```java
static String statement(Invoice invoice, Map<String, Play> plays) {
  Function<Performance,Play> playFor = aPerformance -> plays.get(aPerformance.playID());
  Function<Performance,Integer> amountFor = aPerformance -> {
    var result = 0;
    switch (playFor.apply(aPerformance).type()) {
      case "tragedy" -> {
        result = 40_000;
        if (aPerformance.audience() > 30) result += 1_000 * (aPerformance.audience() - 30);
      }
      case "comedy" -> {
        result = 30_000;
        if (aPerformance.audience() > 20) result += 10_000 + 500 * (aPerformance.audience() - 20);
        result += 300 * aPerformance.audience();
      }
      default -> throw new IllegalArgumentException("unknown type " + playFor.apply(aPerformance).type());
    }
    return result;
  };
  Function<Performance,Integer> volumeCreditsFor = aPerformance -> {
    var result = 0;
    result += Math.max(aPerformance.audience() - 30, 0);
    if ("comedy" == playFor.apply(aPerformance).type()) result += Math.floor(aPerformance.audience() / 5);
    return result;
  };
  Function<Integer,String> usd = aNumber -> {
    final var formatter = NumberFormat.getCurrencyInstance(Locale.US);
    formatter.setCurrency(Currency.getInstance(Locale.US));
    return formatter.format(aNumber);
  };
  Supplier<Integer> totalVolumeCredits = () -> {
    var result = 0;
    for (Performance perf : invoice.performances())
      result += volumeCreditsFor.apply(perf);
    return result;
  };
  Supplier<Integer> totalAmount = () -> {
    var result = 0;
    for (Performance perf : invoice.performances())
      result += amountFor.apply(perf);
    return result;
  };
  var result = "Statement for " + invoice.customer() + "\n";
  for(Performance perf : invoice.performances())
    result += "  " + playFor.apply(perf).name() + ": " + usd.apply(amountFor.apply(perf)/100)
                   + " (" + perf.audience() + " seats)\n";
  result += "Amount owed is " + usd.apply(totalAmount.get()/100) + "\n";
  result += "You earned " + totalVolumeCredits.get() + " credits\n";
  return result;
}
```

So far, my refactoring has focused on adding enough structure to the function so that I can understand it and see it in terms of its logical parts.

This is often the case early in refactoring. Breaking down complicated chunks into small pieces is important, as is naming things well.

Now, I can begin to focus more on the functionality change I want to make—specifically, providing an HTML version of this statement.

In many ways, it's now much easier to do. With all the calculation code split out, all I have to do is write an HTML version of the six lines of code at the bottom.

The problem is that these broken-out functions are nested within the textual statement method, and I don't want to copy and paste them into a new function, however well organized.

Martin Fowler

@martinfowler

Splitting the Phases of Calculation and Formatting

```java
static String statement(Invoice invoice, Map<String, Play> plays) {
  Function<Performance,Play> playFor = aPerformance -> plays.get(aPerformance.playID());
  Function<Performance,Integer> amountFor = aPerformance -> {
    var result = 0;
    switch (playFor.apply(aPerformance).type()) {
      case "tragedy" -> {
        result = 40_000;
        if (aPerformance.audience() > 30) result += 1_000 * (aPerformance.audience() - 30);
      }
      case "comedy" -> {
        result = 30_000;
        if (aPerformance.audience() > 20) result += 10_000 + 500 * (aPerformance.audience() - 20);
        result += 300 * aPerformance.audience();
      }
      default -> throw new IllegalArgumentException("unknown type " + playFor.apply(aPerformance).type());
    }
    return result;
  };
  Function<Performance,Integer> volumeCreditsFor = aPerformance -> {
    var result = 0;
    result += Math.max(aPerformance.audience() - 30, 0);
    if ("comedy" == playFor.apply(aPerformance).type()) result += Math.floor(aPerformance.audience() / 5);
    return result;
  };
  Function<Integer,String> usd = aNumber -> {
    final var formatter = NumberFormat.getCurrencyInstance(Locale.US);
    formatter.setCurrency(Currency.getInstance(Locale.US));
    return formatter.format(aNumber);
  };
  Supplier<Integer> totalVolumeCredits = () -> {
    var result = 0;
    for (Performance perf : invoice.performances())
      result += volumeCreditsFor.apply(perf);
    return result;
  };
  Supplier<Integer> totalAmount = () -> {
    var result = 0;
    for (Performance perf : invoice.performances())
      result += amountFor.apply(perf);
    return result;
  };
  var result = "Statement for " + invoice.customer() + "\n";
  for(Performance perf : invoice.performances())
    result += "  " + playFor.apply(perf).name() + ": " + usd.apply(amountFor.apply(perf)/100)
              + " (" + perf.audience() + " seats)\n";
  result += "Amount owed is " + usd.apply(totalAmount.get()/100) + "\n";
  result += "You earned " + totalVolumeCredits.get() + " credits\n";
  return result;
}
```

I want the same calculation functions to be used by the text and HTML versions of the statement.

There are various ways to do this, but one of my favorite techniques is Split Phase.

My aim here is to divide the logic into two parts: one that calculates the data required for the statement, the other that renders it into text or HTML.

The first phase creates an intermediate data structure that it passes to the second.

I start a Split Phase by applying Extract Function to the code that makes up the second phase.

In this case, that's the statement printing code, which is in fact the entire content of statement.

This, together with all the nested functions, goes into its own top-level function which I call renderPlainText (see next slide).

Martin Fowler
@martinfowler

```
static String statement(Invoice invoice, Map<String, Play> plays) {
  Function<Performance,Play> playFor = perf -> aPerformance.get(aPerformance.playID());
  Function<Performance,Integer> amountFor = aPerformance -> {
    var result = 0;
    switch (playFor.apply(aPerformance).type()) {
      case "tragedy" -> {
        result = 40_000;
        if (aPerformance.audience() > 30) result += 1_000 * (aPerformance.audience() - 30);
      }
      case "comedy" -> {
        result = 30_000;
        if (aPerformance.audience() > 20) result += 10_000 + 500 * (aPerformance.audience() - 20);
        result += 300 * aPerformance.audience();
      }
      default ->
        throw new IllegalArgumentException("unknown type " + playFor.apply(aPerformance).type());
    }
    return result;
  };
  Function<Performance,Integer> volumeCreditsFor = aPerformance -> {
    var result = 0;
    result += Math.max(aPerformance.audience() - 30, 0);
    if ("comedy" == playFor.apply(aPerformance).type())
      result += Math.floor(aPerformance.audience() / 5);
    return result;
  };
  Function<Integer,String> usd = aNumber -> {
    final var formatter = NumberFormat.getCurrencyInstance(Locale.US);
    formatter.setCurrency(Currency.getInstance(Locale.US));
    return formatter.format(aNumber);
  };
  Supplier<Integer> totalVolumeCredits = () -> {
    var result = 0;
    for (Performance perf : invoice.performances())
      result += volumeCreditsFor.apply(perf);
    return result;
  };
  Supplier<Integer> totalAmount = () -> {
    var result = 0;
    for (Performance perf : invoice.performances())
      result += amountFor.apply(perf);
    return result;
  };
  var result = "Statement for " + invoice.customer() + "\n";
  for(Performance perf : invoice.performances())
    result += "  " + playFor.apply(perf).name() + ": " + usd.apply(amountFor.apply(perf)/100)
                   + " (" + perf.audience() + " seats)\n";
  result += "Amount owed is " + usd.apply(totalAmount.get()/100) + "\n";
  result += "You earned " + totalVolumeCredits.get() + " credits\n";
  return result;
}
```

Extract Function renderPlainText

```
static String statement(Invoice invoice, Map<String, Play> plays) {
  return renderPlainText(invoice, plays);
}
static String renderPlainText(Invoice invoice, Map<String, Play> plays) {
  Function<Performance,Play> playFor = aPerformance -> plays.get(aPerformance.playID());
  BiFunction<Performance,Play,Integer> amountFor = aPerformance -> {
    var result = 0;
    switch (playFor.apply(aPerformance).type()) {
      case "tragedy" -> {
        result = 40_000;
        if (aPerformance.audience() > 30) result += 1_000 * (aPerformance.audience() - 30);
      }
      case "comedy" -> {
        result = 30_000;
        if (aPerformance.audience() > 20) result += 10_000 + 500 * (aPerformance.audience() - 20);
        result += 300 * aPerformance.audience();
      }
      default ->
        throw new IllegalArgumentException("unknown type " + playFor.apply(aPerformance).type());
    }
    return result;
  };
  Function<Performance,Integer> volumeCreditsFor = aPerformance -> {
    var result = 0;
    result += Math.max(aPerformance.audience() - 30, 0);
    if ("comedy" == playFor.apply(aPerformance).type())
      result += Math.floor(aPerformance.audience() / 5);
    return result;
  };
  Function<Integer,String> usd = aNumber -> {
    final var formatter = NumberFormat.getCurrencyInstance(Locale.US);
    formatter.setCurrency(Currency.getInstance(Locale.US));
    return formatter.format(aNumber);
  };
  Supplier<Integer> totalVolumeCredits = () -> {
    var result = 0;
    for (Performance perf : invoice.performances())
      result += volumeCreditsFor.apply(perf);
    return result;
  };
  Supplier<Integer> totalAmount = () -> {
    var result = 0;
    for (Performance perf : invoice.performances())
      result += amountFor.apply(perf);
    return result;
  };
  var result = "Statement for " + invoice.customer() + "\n";
  for(Performance perf : invoice.performances())
    result += "  " + playFor.apply(perf).name() + ": " + usd.apply(amountFor.apply(perf)/100)
                   + " (" + perf.audience() + " seats)\n";
  result += "Amount owed is " + usd.apply(totalAmount.get()/100) + "\n";
  result += "You earned " + totalVolumeCredits.get() + " credits\n";
  return result;
}
```

In upcoming slides, **Martin Fowler** will be using the concept of a **Javascript Object**, which he creates and then adds fields to:

```
const foo = {};
foo.bar = abc;
foo.baz = def
```

What we’ll be doing instead in **Java** is introduce a **record**:

```
record Foo(Bar bar, Baz baz)
```

Splitting the Phases of Calculation and Formatting

```
static String statement(Invoice invoice, Map<String, Play> plays) {
  return renderPlainText(invoice, plays);
}
static String renderPlainText(Invoice invoice, Map<String, Play> plays) {
 var result = "Statement for " + invoice.customer() + "\n";
  for(Performance perf : invoice.performances())
    result += "  " + playFor.apply(perf).name() + ": " + usd.apply(amountFor.apply(perf)/100)
                   + " (" + perf.audience() + " seats)\n";
  result += "Amount owed is " + usd.apply(totalAmount.get()/100) + "\n";
  result += "You earned " + totalVolumeCredits.get() + " credits\n";
  return result;
}
```

Martin Fowler
@martinfowler

I do my usual compile-test-commit, then create an object that will act as my intermediate data structure between the two phases. I pass this data object in as an argument to renderPlainText.

```
static String statement(Invoice invoice, Map<String, Play> plays) {
  final var statementData = new StatementData();
  return renderPlainText(statementData, invoice, plays);
}
static String renderPlainText(StatementData data, Invoice invoice, Map<String, Play> plays) {
 var result = "Statement for " + invoice.customer() + "\n";
  for(Performance perf : invoice.performances())
    result += "  " + playFor.apply(perf).name() + ": " + usd.apply(amountFor.apply(perf)/100)
                   + " (" + perf.audience() + " seats)\n";
  result += "Amount owed is " + usd.apply(totalAmount.get()/100) + "\n";
  result += "You earned " + totalVolumeCredits.get() + " credits\n";
  return result;
}
```

```
record StatementData() { }
```

Splitting the Phases of Calculation and Formatting

```java
static String statement(Invoice invoice, Map<String, Play> plays) {
  final var statementData = new StatementData();
  return renderPlainText(statementData, invoice, plays);
}
static String renderPlainText(StatementData data, Invoice invoice, Map<String, Play> plays) {
 var result = "Statement for " + invoice.customer() + "\n";
  for(Performance perf : invoice.performances())
    result += "  " + playFor.apply(perf).name() + ": " + usd.apply(amountFor.apply(perf)/100)
                   + " (" + perf.audience() + " seats)\n";
  result += "Amount owed is " + usd.apply(totalAmount.get()/100) + "\n";
  result += "You earned " + totalVolumeCredits.get() + " credits\n";
  return result;
}
```

```java
record StatementData() { }
```

Martin Fowler

@martinfowler

I now examine the other arguments used by renderPlainText. I want to move the data that comes from them into the intermediate data structure, so that all the calculation code moves into the statement function and renderPlainText operates solely on data passed to it through the data parameter.

My first move is to take the customer and add it to the intermediate object.

```java
static String statement(Invoice invoice, Map<String, Play> plays) {
  final var statementData = new StatementData(invoice .customer());
  return renderPlainText(statementData, invoice, plays);
}
static String renderPlainText(StatementData data, Invoice invoice, Map<String, Play> plays) {
 var result = "Statement for " + data.customer() + "\n";
  for(Performance perf : invoice.performances())
    result += "  " + playFor.apply(perf).name() + ": " + usd.apply(amountFor.apply(perf)/100)
                   + " (" + perf.audience() + " seats)\n";
  result += "Amount owed is " + usd.apply(totalAmount.get()/100) + "\n";
  result += "You earned " + totalVolumeCredits.get() + " credits\n";
  return result;
}
```

```java
record StatementData(String customer) { }
```

Splitting the Phases of Calculation and Formatting

```
record StatementData(String customer) { }
```

```
static String statement(Invoice invoice, Map<String, Play> plays) {
  final var statementData = new StatementData(invoice.customer());
  return renderPlainText(statementData, invoice, plays);
}
static String renderPlainText(StatementData data, Invoice invoice, Map<String, Play> plays) {
 var result = "Statement for " + data.customer() + "\n";
  for(Performance perf : invoice.performances())
    result += "  " + playFor.apply(perf).name() + ": " + usd.apply(amountFor.apply(perf)/100)
               + " (" + perf.audience() + " seats)\n";
  result += "Amount owed is " + usd.apply(totalAmount.get()/100) + "\n";
  result += "You earned " + totalVolumeCredits.get() + " credits\n";
  return result;
}
```

```
Supplier<Integer> totalVolumeCredits = () -> {
  var result = 0;
  for (Performance perf : invoice.performances())
    result += volumeCreditsFor.apply(perf);
  return result;
};
```

```
Supplier<Integer> totalAmount = () -> {
  var result = 0;
  for (Performance perf : invoice.performances())
    result += amountFor.apply(perf);
  return result;
};
```

Similarly, I add the performances, which allows me to delete the invoice parameter to renderPlainText.

Martin Fowler
@martinfowler

```
record StatementData(String customer, List<Performance> performances) { }
```

```
static String statement(Invoice invoice, Map<String, Play> plays) {
  final var statementData = new StatementData(invoice .customer(), invoice.performances());
  return renderPlainText(statementData, plays);
}
static String renderPlainText(StatementData data, Map<String, Play> plays) {
 var result = "Statement for " + data.customer() + "\n";
  for(Performance perf : data.performances())
    result += "  " + playFor.apply(perf).name() + ": " + usd.apply(amountFor.apply(perf)/100)
               + " (" + perf.audience() + " seats)\n";
  result += "Amount owed is " + usd.apply(totalAmount.get()/100) + "\n";
  result += "You earned " + totalVolumeCredits.get() + " credits\n";
  return result;
}
```

```
Supplier<Integer> totalVolumeCredits = () -> {
  var result = 0;
  for (Performance perf : data.performances())
    result += volumeCreditsFor.apply(perf);
  return result;
};
```

```
Supplier<Integer> totalAmount = () -> {
  var result = 0;
  for (Performance perf : data.performances())
    result += amountFor.apply(perf);
  return result;
};
```

@philip_schwarz

In upcoming slides, Martin Fowler introduces the notion of ‘enriching’ Performance objects (during the calculation phase) with additional fields (that are to be used during the formatting phase).

Whilst in Java we’ll ultimately aim to have both a Performance record and an EnrichedPerformance record, we’ll have to start off by ‘enriching’ the Performance record with optional fields, and only later remove the optional fields in favour of a new EnrichedPerformance record.

```
static String statement(Invoice invoice, Map<String, Play> plays) {
  final var statementData = new StatementData(invoice.customer(), invoice.performances());
  return renderPlainText(statementData, plays);
}
static String renderPlainText(StatementData data, Map<String, Play> plays) {
  Function<Performance,Play> playFor = perf -> plays.get(perf.playID());
  Function<Performance,Integer> amountFor = aPerformance -> {
    var result = 0;
    switch (playFor.apply(aPerformance).type()) {
      case "tragedy" -> {
        result = 40_000;
        if (aPerformance.audience() > 30)
          result += 1_000 * (aPerformance.audience() - 30); }
      case "comedy" -> {
        result = 30_000;
        if (aPerformance.audience() > 20)
          result += 10_000 + 500 * (aPerformance.audience() - 20);
        result += 300 * aPerformance.audience(); }
      default -> throw new IllegalArgumentException(
            "unknown type " + playFor.apply(aPerformance).type()); }
    return result;
  };
  Function<Performance,Integer> volumeCreditsFor = aPerformance -> {
    var result = 0;
    result += Math.max(aPerformance.audience() - 30, 0);
    if ("comedy" == playFor.apply(aPerformance).type())
      result += Math.floor(aPerformance.audience() / 5);
    return result;
  };
  var result = "Statement for " + data.customer() + "\n";
  for(Performance perf : data.performances())
    result += "  "+playFor.apply(perf).name()+": "+usd.apply(amountFor.apply(perf)/100)
                    + " (" + perf.audience() + " seats)\n";
  result += "Amount owed is " + usd.apply(totalAmount.get()/100) + "\n";
  result += "You earned " + totalVolumeCredits.get() + " credits\n";
  return result;
}
```

```
static String statement(Invoice invoice, Map<String, Play> plays) {
  Function<Performance,Play> playFor = aPerformance -> plays.get(aPerformance.playID());
  Function<Performance,Performance> enrichPerformance = aPerformance ->
    new Performance(aPerformance.playID(),
                    Optional.of(playFor.apply(aPerformance)),
                    aPerformance.audience());
  final var statementData = new StatementData(
    invoice .customer(),
    invoice.performances().stream().map(enrichPerformance).collect(toList()));
  return renderPlainText(statementData, invoice, plays);
}
static String renderPlainText(StatementData data, Map<String, Play> plays) {
  Function<Performance,Integer> amountFor = aPerformance -> {
    var result = 0;
    switch (aPerformance.play().get().type()) {
      case "tragedy" -> {
        result = 40_000;
        if (aPerformance.audience() > 30)
          result += 1_000 * (aPerformance.audience() - 30); }
      case "comedy" -> {
        result = 30_000;
        if (aPerformance.audience() > 20)
          result += 10_000 + 500 * (aPerformance.audience() - 20);
        result += 300 * aPerformance.audience(); }
      default -> throw new IllegalArgumentException(
            "unknown type " + aPerformance.play().get().type()); }
    return result;
  };
  Function<Performance,Integer> volumeCreditsFor = aPerformance -> {
    var result = 0;
    result += Math.max(aPerformance.audience() - 30, 0);
    if ("comedy" == aPerformance.play().get().type())
      result += Math.floor(aPerformance.audience() / 5);
    return result;
  };
  var result = "Statement for " + data.customer() + "\n";
  for(Performance perf : data.performances())
    result += "  "+ perf.play().get().name()+": "+usd.apply(amountFor.apply(perf)/100)
                    + " (" + perf.audience() + " seats)\n";
  result += "Amount owed is " + usd.apply(totalAmount.get()/100) + "\n";
  result += "You earned " + totalVolumeCredits.get() + " credits\n";
  return result;
}
```

Now I'd like the play name to come from the intermediate data. To do this, I need to enrich the performance record with data from the play.

```
record Performance(String playID, int audience) { }
```

```
record Performance(String playID, Optional<Play> play, int audience) {
  Performance(String playID, int audience) { this(playID, Optional.empty(), audience); }}
```

Martin Fowler @martinfowler

```
static String statement(Invoice invoice, Map<String, Play> plays) {
  Function<Performance,Play> playFor = aPerformance -> plays.get(aPerformance.playID());
  Function<Performance,Performance> enrichPerformance = aPerformance ->
    new Performance(aPerformance.playID(),
                    Optional.of(playFor.apply(aPerformance)),
                    aPerformance.audience());
  final var statementData = new StatementData(
    invoice .customer(),
    invoice.performances().stream().map(enrichPerformance).collect(toList()));
  return renderPlainText(statementData, plays);
}
static String renderPlainText(StatementData data, Map<String, Play> plays) {
  Function<Performance,Integer> amountFor = aPerformance -> {
    var result = 0;
    switch (aPerformance.play().get().type()) {
      case "tragedy" -> {
        result = 40_000;
        if (aPerformance.audience() > 30)
          result += 1_000 * (aPerformance.audience() - 30); }
      case "comedy" -> {
        result = 30_000;
        if (aPerformance.audience() > 20)
          result += 10_000 + 500 * (aPerformance.audience() - 20);
        result += 300 * aPerformance.audience(); }
      default -> throw new IllegalArgumentException(
         "unknown type " + aPerformance.play().get().type()); }
    return result;
  };
  Supplier<Integer> totalAmount = () -> {
    var result = 0;
    for (Performance perf : data.performances())
      result += amountFor.apply(perf);
    return result;
  };
var result = "Statement for " + data.customer() + "\n";
  for(Performance perf : data.performances())
    result += "  "+ perf.play().get().name()+": "+usd.apply(amountFor.apply(perf)/100)
                + " (" + perf.audience() + " seats)\n";
  result += "Amount owed is " + usd.apply(totalAmount.get()/100) + "\n";
  result += "You earned " + totalVolumeCredits.get() + " credits\n";
  return result;
}
```

I then move amountFor in a similar way.

Martin Fowler

```
record Performance(
  String playID, Optional<Play> play, int audience
){ Performance(String playID, int audience) {
    this(playID, Optional.empty(), audience); }}
```

```
static String statement(Invoice invoice, Map<String, Play> plays) {
  Function<Performance,Play> playFor = aPerformance -> plays.get(aPerformance.playID());
  Function<Performance,Integer> amountFor = aPerformance -> {
    var result = 0;
    switch (playFor.apply(aPerformance).type()) {
      case "tragedy" -> {
        result = 40_000;
        if (aPerformance.audience() > 30)
          result += 1_000 * (aPerformance.audience() - 30); }
      case "comedy" -> {
        result = 30_000;
        if (aPerformance.audience() > 20)
          result += 10_000 + 500 * (aPerformance.audience() - 20);
        result += 300 * aPerformance.audience(); }
      default -> throw new IllegalArgumentException(
         "unknown type " + playFor.apply(aPerformance).type()); }
    return result;
  };
  Function<Performance,Performance> enrichPerformance = aPerformance ->
    new Performance(aPerformance.playID(),
                    Optional.of(playFor.apply(aPerformance)),
                    aPerformance.audience(),
                    Optional.of(amountFor.apply(aPerformance)));
  final var statementData = new StatementData(
    invoice .customer(),
    invoice.performances().stream().map(enrichPerformance).collect(toList()));
  return renderPlainText(statementData);
}
static String renderPlainText(StatementData data) {
  Supplier<Integer> totalAmount = () -> {
    var result = 0;
    for (Performance perf : data.performances())
      result += perf.amount().get();
    return result;
  };
  var result = "Statement for " + data.customer() + "\n";
  for(Performance perf : data.performances())
    result += "  "+ perf.play().get().name()+": "+usd.apply(perf.amount().get()/100)
                + " (" + perf.audience() + " seats)\n";
  result += "Amount owed is " + usd.apply(totalAmount.get()/100) + "\n";
  result += "You earned " + totalVolumeCredits.get() + " credits\n";
  return result;
}
```

```
record Performance(
  String playID, Optional<Play> play, int audience, Optional<Integer> amount
){ Performance(String playID, int audience) {
    this(playID, Optional.empty(), audience, Optional.empty()); }}
```

Note that, on the previous slide, I have already removed the **plays** parameter of **renderPlainText**, since it is no longer used. In the book, this doesn’t happen till later in this section.

Splitting the Phases of Calculation and Formatting

```
static String statement(Invoice invoice, Map<String, Play> plays) {

  Function<Performance,Performance> enrichPerformance = aPerformance ->
    new Performance(aPerformance.playID(),
                    Optional.of(playFor.apply(aPerformance)),
                    aPerformance.audience(),
                    Optional.of(amountFor.apply(aPerformance)));
  final var statementData = new StatementData(
    invoice .customer(),
    invoice.performances().stream().map(enrichPerformance).collect(toList()));
  return renderPlainText(statementData);
}

static String renderPlainText(StatementData data) {

  Function<Performance,Integer> volumeCreditsFor = aPerformance -> {
    var result = 0;
    result += Math.max(aPerformance.audience() - 30, 0);
    if ("comedy" == aPerformance.play().get().type())
      result += Math.floor(aPerformance.audience() / 5);
    return result;
  };

  Supplier<Integer> totalVolumeCredits = () -> {
    var result = 0;
    for (Performance perf : data.performances())
      result += volumeCreditsFor.apply(perf);
    return result;
  };
  var result = "Statement for " + data.customer() + "\n";
  for(Performance perf : data.performances())
    result += "  "+ perf.play().get().name()+": "+usd.apply(perf.amount().get()/100)
                  + " (" + perf.audience() + " seats)\n";
  result += "Amount owed is " + usd.apply(totalAmount.get()/100) + "\n";
  result += "You earned " + totalVolumeCredits.get() + " credits\n";
  return result;
}
```

Next, I move the volumeCreditsFor calculation.

Martin Fowler

@martinfowler

```
record Performance(
  String playID, Optional<Play> play, int audience,
  Optional<Integer> amount
){ Performance(String playID, int audience) {
    this(playID, Optional.empty(), audience, Optional.empty()); }}
```

```
static String statement(Invoice invoice, Map<String, Play> plays) {

  Function<Performance,Integer> volumeCreditsFor = aPerformance -> {
    var result = 0;
    result += Math.max(aPerformance.audience() - 30, 0);
    if ("comedy" == playFor.apply(aPerformance).type())
      result += Math.floor(aPerformance.audience() / 5);
    return result;
  };

  Function<Performance,Performance> enrichPerformance = aPerformance ->
    new Performance(aPerformance.playID(),
                    Optional.of(playFor.apply(aPerformance)),
                    aPerformance.audience(),
                    Optional.of(amountFor.apply(aPerformance)),
                    Optional.of(volumeCreditsFor.apply(aPerformance)));
  final var statementData = new StatementData(
    invoice .customer(),
    invoice.performances().stream().map(enrichPerformance).collect(toList()));
  return renderPlainText(statementData);
}

static String renderPlainText(StatementData data) {

  Supplier<Integer> totalVolumeCredits = () -> {
    var result = 0;
    for (Performance perf : data.performances())
      result += perf.volumeCredits().get();
    return result;
  };
  var result = "Statement for " + data.customer() + "\n";
  for(Performance perf : data.performances())
    result += "  "+ perf.play().get().name()+": "+usd.apply(perf.amount().get()/100)
                  + " (" + perf.audience() + " seats)\n";
  result += "Amount owed is " + usd.apply(totalAmount.get()/100) + "\n";
  result += "You earned " + totalVolumeCredits.get() + " credits\n";
  return result;
}
```

```
record Performance(
  String playID, Optional<Play> play, int audience,
  Optional<Integer> amount, Optional<Integer> volumeCredits
){ Performance(String playID, int audience) {
    this(playID, Optional.empty(), audience, Optional.empty(), Optional.empty()); }}
```

We can now remove the optional Performance fields by introducing an EnrichedPerformance.

Splitting the Phases of Calculation and Formatting

```
static String statement(Invoice invoice, Map<String, Play> plays) {
  Function<Performance,Performance> enrichPerformance = aPerformance ->
    new Performance(
      aPerformance.playID(),
      Optional.of(playFor.apply(aPerformance)),
      aPerformance.audience(),
      Optional.of(amountFor.apply(aPerformance)),
      Optional.of(volumeCreditsFor.apply(aPerformance)));
  final var statementData = new StatementData(
    invoice .customer(),
    invoice.performances().stream().map(enrichPerformance).collect(toList()));
  return renderPlainText(statementData);
}

static String renderPlainText(StatementData data) {
  Supplier<Integer> totalVolumeCredits = () -> {
    var result = 0;
    for (Performance perf : data.performances())
      result += perf.volumeCredits().get();
    return result;
  };
  Supplier<Integer> totalAmount = () -> {
    var result = 0;
    for (Performance perf : data.performances())
      result += perf.amount().get();
    return result;
  };
  var result = "Statement for " + data.customer() + "\n";
  for(Performance perf : data.performances())
    result += "  "+ perf.play().get().name()+": "+usd.apply(perf.amount().get()/100)
                + " (" + perf.audience() + " seats)\n";
  result += "Amount owed is " + usd.apply(totalAmount.get()/100) + "\n";
  result += "You earned " + totalVolumeCredits.get() + " credits\n";
  return result;
}
```

```
record Performance(
  String playID,
  Optional<Play> play,
  int audience,
  Optional<Integer> amount,
  Optional<Integer> volumeCredits
){ Performance(
     String playID, int audience)
{
  this(playID,
       Optional.empty(),
       audience,
       Optional.empty(),
       Optional.empty()); }}
```

```
record StatementData(String customer, List<Performance> performances){ }
```

```
static String statement(Invoice invoice, Map<String, Play> plays) {
  Function<Performance,EnrichedPerformance> enrichPerformance = aPerformance ->
    new EnrichedPerformance(
      aPerformance.playID(),
      playFor.apply(aPerformance),
      aPerformance.audience(),
      amountFor.apply(aPerformance),
      volumeCreditsFor.apply(aPerformance));
  final var statementData = new StatementData(
    invoice .customer(),
    invoice.performances().stream().map(enrichPerformance).collect(toList()));
  return renderPlainText(statementData);
}

static String renderPlainText(StatementData data) {
  Supplier<Integer> totalVolumeCredits = () -> {
    var result = 0;
    for (EnrichedPerformance perf : data.performances())
      result += perf.volumeCredits();
    return result;
  };
  Supplier<Integer> totalAmount = () -> {
    var result = 0;
    for (EnrichedPerformance perf : data.performances())
      result += perf.amount();
    return result;
  };
  var result = "Statement for " + data.customer() + "\n";
  for(EnrichedPerformance perf : data.performances())
    result += "  "+ perf.play().name()+": "+usd.apply(perf.amount()/100)
                + " (" + perf.audience() + " seats)\n";
  result += "Amount owed is " + usd.apply(totalAmount.get()/100) + "\n";
  result += "You earned " + totalVolumeCredits.get() + " credits\n";
  return result;
}
```

```
record Performance(
  String playID,
  int audience
) { }
```

```
record EnrichedPerformance(
  String playID,
  Play play,
  int audience,
  Integer amount,
  Integer volumeCredits
) { }
```

```
record StatementData(String customer, List<EnrichedPerformance> performances) { }
```

Splitting the Phases of Calculation and Formatting

```java
static String statement(Invoice invoice, Map<String, Play> plays) {
  Function<Performance,EnrichedPerformance> enrichPerformance = aPerformance ->
    new EnrichedPerformance(
      aPerformance.playID(),
      playFor.apply(aPerformance),
      aPerformance.audience(),
      amountFor.apply(aPerformance),
      volumeCreditsFor.apply(aPerformance));
  final var statementData = new StatementData(
    invoice .customer(),
    invoice.performances().stream().map(enrichPerformance).collect(toList()));
  return renderPlainText(statementData);
}

static String renderPlainText(StatementData data) {
  Supplier<Integer> totalVolumeCredits = () -> {
    var result = 0;
    for (EnrichedPerformance perf : data.performances())
      result += perf.volumeCredits();
    return result;
  };
  Supplier<Integer> totalAmount = () -> {
    var result = 0;
    for (EnrichedPerformance perf : data.performances())
      result += perf.amount();
    return result;
  };
  var result = "Statement for " + data.customer() + "\n";
  for(EnrichedPerformance perf : data.performances())
    result += "  "+ perf.play().name()+": "+usd.apply(perf.amount()/100)
            + " (" + perf.audience() + " seats)\n";
  result += "Amount owed is " + usd.apply(totalAmount.get()/100) + "\n";
  result += "You earned " + totalVolumeCredits.get() + " credits\n";
  return result;
}
```

Finally, I move the two calculations of the totals.

Martin Fowler

@martinfowler

```java
record StatementData(
  String customer,
  List<EnrichedPerformance> performances
) { }
```

```java
static String statement(Invoice invoice, Map<String, Play> plays) {
  Function<List<EnrichedPerformance>,Integer> totalVolumeCredits = performances -> {
    var result = 0;
    for (EnrichedPerformance perf : performances)
      result += perf.volumeCredits();
    return result;
  };
  Function<List<EnrichedPerformance>,Integer> totalAmount = performances -> {
    var result = 0;
    for (EnrichedPerformance perf : performances)
      result += perf.amount();
    return result;
  };
  Function<Performance,EnrichedPerformance> enrichPerformance = aPerformance ->
    new EnrichedPerformance(
      aPerformance.playID(),
      playFor.apply(aPerformance),
      aPerformance.audience(),
      amountFor.apply(aPerformance),
      volumeCreditsFor.apply(aPerformance));
  final var enrichedPerformances =
    invoice.performances().stream().map(enrichPerformance).collect(toList());
  final var statementData = new StatementData(
    invoice .customer(),
    enrichedPerformances,
    totalAmount.apply(enrichedPerformances),
    totalVolumeCredits.apply(enrichedPerformances));
  return renderPlainText(statementData);
}

static String renderPlainText(StatementData data) {
var result = "Statement for " + data.customer() + "\n";
  for(EnrichedPerformance perf : data.performances())
    result += "  "+ perf.play().name()+": "+usd.apply(perf.amount()/100)
            + " (" + perf.audience() + " seats)\n";
  result += "Amount owed is " + usd.apply(data.totalAmount()/100) + "\n";
  result += "You earned " + data.totalVolumeCredits() + " credits\n";
  return result;
}
```

```java
record StatementData(
  String customer,
  List<EnrichedPerformance> performances,
  Integer totalAmount,
  Integer totalVolumeCredits) { }
```

Splitting the Phases of Calculation and Formatting

```java
Function<List<EnrichedPerformance>,Integer> totalVolumeCredits = performances -> {
  var result = 0;
  for (EnrichedPerformance perf : performances)
    result += perf.volumeCredits();
  return result;
};
Function<List<EnrichedPerformance>,Integer> totalAmount = performances -> {
  var result = 0;
  for (EnrichedPerformance perf : performances)
    result += perf.amount();
  return result;
};
```

Martin Fowler

@martinfowler

I can't resist a couple quick shots of **Remove Loop with Pipeline**

```java
Function<List<EnrichedPerformance>,Integer> totalVolumeCredits = performances ->
  performances.stream().collect(reducing(0,EnhancedPerformance::volumeCredits,Integer::sum));

Function<List<EnrichedPerformance>,Integer> totalAmount = performances ->
  performances.stream().collect(reducing(0,EnhancedPerformance::amount,Integer::sum));
```

Splitting the Phases of Calculation and Formatting

```java
static String statement(Invoice invoice, Map<String, Play> plays) {
  final var enrichedPerformances = invoice.performances().stream().map(enrichPerformance).collect(toList());
  final var statementData = new StatementData(
    invoice .customer(),
    enrichedPerformances,
    totalAmount.apply(enrichedPerformances),
    totalVolumeCredits.apply(enrichedPerformances));
  return renderPlainText(statementData);
}
```

Martin Fowler

@martinfowler

I now extract all the **first-phase code** into its own function.

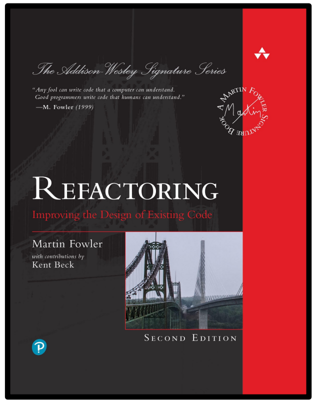
```java
static String statement(Invoice invoice, Map<String, Play> plays) {
  return renderPlainText(createStatementData(invoice,plays));
}

static StatementData createStatementData(Invoice invoice, Map<String, Play> plays) {
  final var enrichedPerformances = invoice.performances().stream().map(enrichPerformance).collect(toList());
  return new StatementData(
    invoice.customer(),
    enrichedPerformances,
    totalAmount.apply(enrichedPerformances),
    totalVolumeCredits.apply(enrichedPerformances));
}
```

Note that on the previous slide, when we extracted **createStatementData**, all the functions nested inside **statement**, e.g. **totalAmount** and **totalVolumeCredits**, also moved along and are now nested in **createStatementData**.

Martin Fowler

@martinfowler

- Decomposing the `statement` Function
  - Removing the play Variable
  - Extracting Volume Credits
  - Removing the `formatter` Variable
  - Removing Total Volume Credits
  - Removing Total Amount
  - Status: Lots of Nested Functions
  - Splitting the Phases of Calculation and Formatting
  - Status: Separated into Two Files (and Phases)

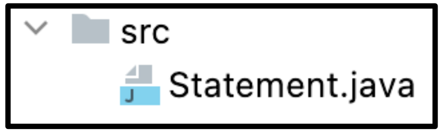

The Addison-Wesley Signature Series
Refactoring
Improving the Design of Existing Code
Martin Fowler
with contributions by
Kent Beck
Second Edition

Status: Separated into Two Files (and Phases)

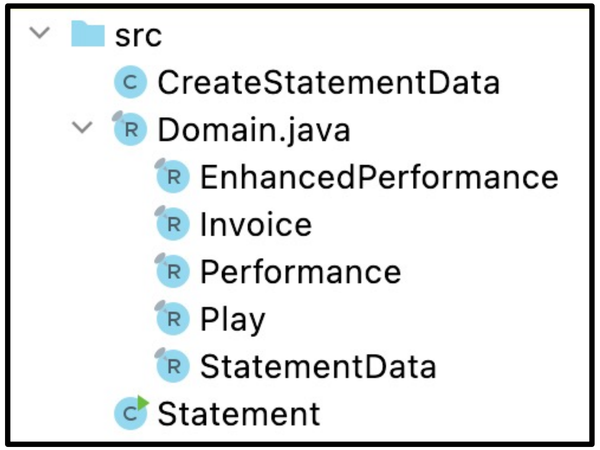
Martin Fowler
@martinfowler

Since **createStatementData** is now clearly separate, I move it into its own file.

See next slide – I also moved the **records** into their own file.



```
src
    Statement.java
```



```
src
    CreateStatementData
    Domain.java
        EnhancedPerformance
        Invoice
        Performance
        Play
        StatementData
    Statement
```

Status: Separated into Two Files (and Phases)

```
public class Statement { … }
```

Statement.java

```
static String statement(Invoice invoice, Map<String, Play> plays) {
  return renderPlainText(
    CreateStatementData.createStatementData(invoice,plays));
}

static String renderPlainText(StatementData data) {

  Function<Integer,String> usd = aNumber -> {
    final var formatter = NumberFormat.getCurrencyInstance(Locale.US);
    formatter.setCurrency(Currency.getInstance(Locale.US));
    return formatter.format(aNumber);
  };

  var result = "Statement for " + data.customer() + "\n";
  for(EnrichedPerformance perf : data.performances())
    result += "  "+ perf.play().name()+": "+usd.apply(perf.amount()/100)
                  + " (" + perf.audience() + " seats)\n";
  result += "Amount owed is " + usd.apply(data.totalAmount()/100) + "\n";
  result += "You earned " + data.totalVolumeCredits() + " credits\n";
  return result;
}
```

Domain.java

```
record Performance(String playID, int audience) { }

record EnrichedPerformance(
  String playID,
  Play play,
  int audience,
  Integer amount,
  Integer volumeCredits
) { }

record Invoice(String customer, List<Performance> performances) { }

record Play(String name, String type) { }

record StatementData(
  String customer,
  List<EnrichedPerformance> performances,
  Integer totalAmount,
  Integer totalVolumeCredits) { }
```

```
public class CreateStatementData { … }
```

CreateStatementData.java

```
static StatementData createStatementData(Invoice invoice, Map<String, Play> plays) {

  Function<Performance,Play> playFor = aPerformance -> plays.get(aPerformance.playID());

  Function<Performance,Integer> amountFor = aPerformance -> {
    var result = 0;
    switch (playFor.apply(aPerformance).type()) {
      case "tragedy" -> {
        result = 40_000;
        if (aPerformance.audience() > 30)
          result += 1_000 * (aPerformance.audience() - 30); }
      case "comedy" -> {
        result = 30_000;
        if (aPerformance.audience() > 20)
          result += 10_000 + 500 * (aPerformance.audience() - 20);
        result += 300 * aPerformance.audience(); }
      default -> throw new IllegalArgumentException(
            "unknown type " + playFor.apply(aPerformance).type()); }
    return result; };

  Function<Performance,Integer> volumeCreditsFor = aPerformance -> {
    var result = 0;
    result += Math.max(aPerformance.audience() - 30, 0);
    if ("comedy" == playFor.apply(aPerformance).type())
      result += Math.floor(aPerformance.audience() / 5);
    return result; };

  Function<List<EnhancedPerformance>,Integer> totalVolumeCredits = (performances) ->
    performances.stream().collect(
      reducing(0,EnhancedPerformance::volumeCredits,Integer::sum));

  Function<List<EnhancedPerformance>,Integer> totalAmount = (performances) ->
    performances.stream().collect(
      reducing(0,EnhancedPerformance::amount,Integer::sum));

  Function<Performance,EnrichedPerformance> enrichPerformance = aPerformance ->
    new EnrichedPerformance(
      aPerformance.playID(),
      playFor.apply(aPerformance),
      aPerformance.audience(),
      amountFor.apply(aPerformance),
      volumeCreditsFor.apply(aPerformance));

  final var enrichedPerformances =
    invoice.performances().stream().map(enrichPerformance).collect(toList());
  return new StatementData(
    invoice.customer(),
    enrichedPerformances,
    totalAmount.apply(enrichedPerformances),
    totalVolumeCredits.apply(enrichedPerformances));
}
```

Status: Separated into Two Files (and Phases)

Martin Fowler

@martinfowler

It is now easy to write an **HTML** version of **statement** and **renderPlainText** (I moved **usd** to the top level so that **renderHtml** could use it).

```
static String htmlStatement(Invoice invoice, Map<String, Play> plays) {
  return renderHtml(CreateStatementData.createStatementData(invoice, plays));
}

static String renderHtml(StatementData data) {
  var result = "<h1>Statement for ” + data.customer() + "</h1>\n";
  result += "<table>\n";
  result += "<tr><th>play</th><th>seats</th><th>cost</th></tr>\n";
  for (EnrichedPerformance perf : data.performances()) {
    result += "<tr><td>" + perf.play().name() + "</td><td>" + perf.audience() + "</td>";
    result += "<td>" + usd(perf.amount() / 100) + "</td></tr>\n";
  }
  result += "</table>\n";
  result += "<p>Amount owed is <em>" + usd(data.totalAmount()/100) + "</em></p>\n";
  result += "<p>You earned <em>” + data.totalVolumeCredits() + "</em> credits</p>\n";
  return result;
}
```

Status: Separated into Two Files (and Phases)

Let’s add an assertion test for **htmlStatement**.

@philip_schwarz

```
public static void main(String[] args) {

 if (!Statement.statement(invoices.get(0), plays).equals(
  """
  Statement for BigCo
   Hamlet: $650.00 (55 seats)
   As You Like It: $580.00 (35 seats)
   Othello: $500.00 (40 seats)
  Amount owed is $1,730.00
  You earned 47 credits
  """
 )) throw new AssertionError();

 if (!Statement.htmlStatement(invoices.get(0), plays).equals(
  """
  <h1>Statement for BigCo</h1>
  <table>
  <tr><th>play</th><th>seats</th><th>cost</th></tr>
  <tr><td>Hamlet</td><td>55</td><td>$650.00</td></tr>
  <tr><td>As You Like It</td><td>35</td><td>$580.00</td></tr>
  <tr><td>Othello</td><td>40</td><td>$500.00</td></tr>
  </table>
  <p>Amount owed is <em>$1,730.00</em></p>
  <p>You earned <em>47</em> credits</p>
  """
 )) throw new AssertionError();
}
```

```
static final List<Invoice> invoices =
 List.of(
  new Invoice(
   "BigCo",
   List.of(new Performance( "hamlet", 55),
           new Performance("as-like", 35),
           new Performance("othello", 40))));

static final Map<String,Play> plays =
 Map.of("hamlet" , new Play("Hamlet", "tragedy"),
        "as-like", new Play("As You Like It", "comedy"),
        "othello", new Play("Othello", "tragedy"));
```

```
Function<Integer,String> usd = aNumber -> {
  final var formatter = NumberFormat.getCurrencyInstance(Locale.US);
  formatter.setCurrency(Currency.getInstance(Locale.US));
  return formatter.format(aNumber);
};
```

Rather than being nested inside the **statement** function, the **usd** function is now at the top level, so that it can be used by both **renderPlainText** and **renderHtml**, so it no longer needs to be a **lambda function**.

```
static String usd(int aNumber) {
  final var formatter = NumberFormat.getCurrencyInstance(Locale.US);
  formatter.setCurrency(Currency.getInstance(Locale.US));
  return formatter.format(aNumber);
}
```

Martin Fowler

@martinfowler

There are more things I could do to simplify the printing logic, but this will do for the moment.

I always have to strike a balance between all the refactorings I could do and adding new features.

At the moment, most people under-prioritize refactoring—but there still is a **balance**.

My rule is a variation on the camping rule:

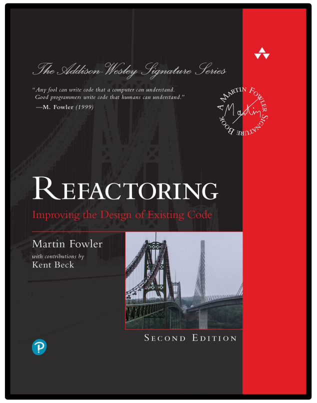
Always leave the code base healthier than when you found it.

It will never be perfect, but it should be better.

Martin Fowler

@martinfowler

- Decomposing the `statement` Function
  - Removing the play Variable
  - Extracting Volume Credits
  - Removing the `formatter` Variable
  - Removing Total Volume Credits
  - Removing Total Amount
  - Status: Lots of Nested Functions
  - Splitting the Phases of Calculation and Formatting
  - Status: Separated into Two Files (and Phases)
  - Reorganising the Calculations by Type


The Addison-Wesley Signature Series
Refactoring
Improving the Design of Existing Code
Martin Fowler
with contributions by
Kent Beck
Second Edition

Martin Fowler

@martinfowler

Now I'll turn my attention to the next feature change: supporting more categories of plays, each with its own charging and volume credits calculations. At the moment, to make changes here I have to go into the calculation functions and edit the conditions in there.

The amountFor function highlights the central role the type of play has in the choice of calculations—but conditional logic like this tends to decay as further modifications are made unless it's reinforced by more structural elements of the programming language.

There are various ways to introduce structure to make this explicit, but in this case a natural approach is type polymorphism—a prominent feature of classical object-orientation. Classical OO has long been a controversial feature in the JavaScript world, but the ECMAScript 2015 version provides a sound syntax and structure for it. So it makes sense to use it in a right situation—like this one.

My overall plan is to set up an inheritance hierarchy with comedy and tragedy subclasses that contain the calculation logic for those cases. Callers call a polymorphic amount function that the language will dispatch to the different calculations for the comedies and tragedies. I'll make a similar structure for the volume credits calculation. To do this, I utilize a couple of refactorings.

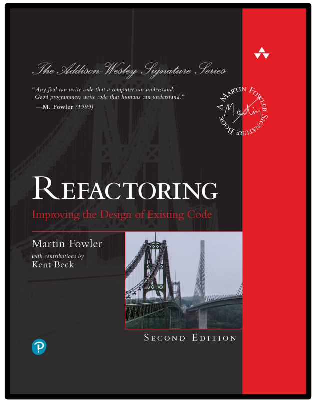
The core refactoring is Replace Conditional with Polymorphism, which changes a hunk of conditional code with polymorphism. But before I can do Replace Conditional with Polymorphism, I need to create an inheritance structure of some kind. I need to create a class to host the amount and volume credit functions.

```
Function<Performance,Integer> amountFor = aPerformance -> {
  var result = 0;
  switch (playFor.apply(aPerformance).type()) {
    case "tragedy" -> {
      result = 40_000;
      if (aPerformance.audience() > 30)
        result += 1_000 * (aPerformance.audience() - 30); }
    case "comedy" -> {
      result = 30_000;
      if (aPerformance.audience() > 20)
        result += 10_000 + 500 * (aPerformance.audience() - 20);
      result += 300 * aPerformance.audience(); }
    default -> throw new IllegalArgumentException(
          "unknown type " + playFor.apply(aPerformance).type());
  }
  return result;
};
```

Martin Fowler

@martinfowler

- Decomposing the `statement` Function
  - Removing the play Variable
  - Extracting Volume Credits
  - Removing the `formatter` Variable
  - Removing Total Volume Credits
  - Removing Total Amount
  - Status: Lots of Nested Functions
  - Splitting the Phases of Calculation and Formatting
  - Status: Separated into Two Files (and Phases)
  - Reorganising the Calculations by Type
    - Creating a Performance Calculator


The Addison-Wesley Signature Series
REFACTORING
Improving the Design of Existing Code
Martin Fowler
with contributions by
Kent Beck
SECOND EDITION

Creating a Performance Calculator

```
Function<Performance,EnrichedPerformance> enrichPerformance = aPerformance ->
  new EnrichedPerformance(
    aPerformance.playID(),
    playFor.apply(aPerformance),
    aPerformance.audience(),
    amountFor.apply(aPerformance),
    volumeCreditsFor.apply(aPerformance));
```

Martin Fowler
@martinfowler

The **enrichPerformance** function is the key, since it populates the **intermediate data structure** with the data for each **performance**.

Currently, it calls the conditional functions for **amount** and **volume credits**. What I need it to do is call those functions on a **host class**.

Since that class **hosts** functions for calculating data about **performances**, I'll call it a **performance calculator**.

```
record PerformanceCalculator(Performance performance) { }
```

```
Function<Performance,EnrichedPerformance> enrichPerformance = aPerformance -> {
  final var calculator = new PerformanceCalculator(aPerformance);
  return new EnrichedPerformance(
    aPerformance.playID(),
    playFor.apply(aPerformance),
    aPerformance.audience(),
    amountFor.apply(aPerformance),
    volumeCreditsFor.apply(aPerformance));
}
```

Creating a Performance Calculator

```
record PerformanceCalculator(Performance performance) { }
```

```
Function<Performance,EnrichedPerformance> enrichPerformance =
aPerformance -> {
  var calculator = new PerformanceCalculator(aPerformance);
  new EnrichedPerformance(
    aPerformance.playID(),
    playFor.apply(aPerformance),
    aPerformance.audience(),
    amountFor.apply(aPerformance),
    volumeCreditsFor.apply(aPerformance));
}
```

Martin Fowler
@martinfowler

So far, this new object isn't doing anything. I want to move behavior into it—and I'd like to start with the simplest thing to move, which is the play record.

Strictly, I don't need to do this, as it's not varying polymorphically, but this way I'll keep all the data transforms in one place, and that consistency will make the code clearer.

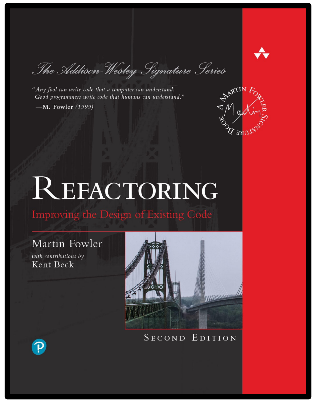
```
record PerformanceCalculator(Performance performance, Play play) { }
```

```
Function<Performance,EnrichedPerformance> enrichPerformance = aPerformance -> {
  var calculator = new PerformanceCalculator(aPerformance, playFor.apply(aPerformance));
  new EnrichedPerformance(
    aPerformance.playID(),
    calculator.play(),
    aPerformance.audience(),
    amountFor.apply(aPerformance),
    volumeCreditsFor.apply(aPerformance));
}
```

Martin Fowler

@martinfowler

- Decomposing the `statement` Function
  - Removing the play Variable
  - Extracting Volume Credits
  - Removing the `formatter` Variable
  - Removing Total Volume Credits
  - Removing Total Amount
  - Status: Lots of Nested Functions
  - Splitting the Phases of Calculation and Formatting
  - Status: Separated into Two Files (and Phases)
  - Reorganising the Calculations by Type
    - Creating a Performance Calculator
    - Moving Functions into the Calculator


The Addison-Wesley Signature Series
Refactoring
Improving the Design of Existing Code
Martin Fowler
with contributions by
Kent Beck
Second Edition

Moving Functions into the Calculator

```
record PerformanceCalculator(Performance performance, Play play) { }
```

```
Function<Performance,Integer> amountFor = aPerformance -> {
  var result = 0;
  switch (playFor.apply(aPerformance).type()) {
    case "tragedy" -> {
      result = 40_000;
      if (aPerformance.audience() > 30)
        result += 1_000 * (aPerformance.audience() - 30); }
    case "comedy" -> {
      result = 30_000;
      if (aPerformance.audience() > 20)
        result += 10_000 + 500 * (aPerformance.audience() - 20);
      result += 300 * aPerformance.audience(); }
    default -> throw new IllegalArgumentException(
          "unknown type " + playFor.apply(aPerformance).type());
  }
  return result;
};
```

The next bit of **logic** I move is rather more substantial for calculating the **amount** for a **performance**…

The first part of this refactoring is to copy the logic over to its new context—the calculator class.

Then, I adjust the code to fit into its new home, changing aPerformance to performance and playFor(aPerformance) to play.

Martin Fowler
@martinfowler

Move Function amountFor (copy logic)

```
record PerformanceCalculator(Performance performance, Play play) {
  int amount() {
    var result = 0;
    switch (play.type()) {
      case "tragedy" -> {
        result = 40_000;
        if (performance.audience() > 30)
          result += 1_000 * (performance.audience() - 30); }
      case "comedy" -> {
        result = 30_000;
        if (performance.audience() > 20)
          result += 10_000 + 500 * (performance.audience() - 20);
        result += 300 * performance.audience(); }
      default -> throw new IllegalArgumentException(
            "unknown type " + play.type());
    }
    return result;
  }
}
```

Moving Functions into the Calculator

```
Function<Performance,Integer> amountFor = aPerformance -> {
  var result = 0;
  switch (playFor.apply(aPerformance).type()) {
    case "tragedy" -> {
      result = 40_000;
      if (aPerformance.audience() > 30)
        result += 1_000 * (aPerformance.audience() - 30); }
    case "comedy" -> {
      result = 30_000;
      if (aPerformance.audience() > 20)
        result += 10_000 + 500 * (aPerformance.audience() - 20);
      result += 300 * aPerformance.audience(); }
    default -> throw new IllegalArgumentException(
          "unknown type " + playFor.apply(aPerformance).type());
  }
  return result;
};
```

Martin Fowler

@martinfowler

Once the new function fits its home, I take the original function and turn it into a **delegating function** so it calls the new function.

Move Function amountFor (delegation)

```
Function<Performance,Integer> amountFor = aPerformance ->
  new PerformanceCalculator(aPerformance, playFor.apply(aPerformance)).amount();
```

Moving Functions into the Calculator

```
Function<Performance,Integer> amountFor = aPerformance ->
  new PerformanceCalculator(aPerformance, playFor.apply(aPerformance)).amount();
```

```
Function<Performance,EnrichedPerformance> enrichPerformance = aPerformance -> {
  var calculator = new PerformanceCalculator(aPerformance, playFor.apply(aPerformance));
  return new EnrichedPerformance(
    aPerformance.playID(),
    calculator.play(),
    aPerformance.audience(),
    amountFor.apply(aPerformance),
    volumeCreditsFor.apply(aPerformance));
}
```

Martin Fowler
@martinfowler

With that done, I use **Inline Function** to call the new **amount** function directly.

Yes, we are not just inlining **amountFor**, we are then taking into consideration the fact that the body of **amountFor** that we have just inlined is equivalent to the simpler expression **calculator.amount**.

Inline Function amountFor

```
Function<Performance,EnrichedPerformance> enrichPerformance = aPerformance -> {
  var calculator = new PerformanceCalculator(aPerformance, playFor.apply(aPerformance));
  return new EnrichedPerformance(
    aPerformance.playID(),
    calculator.play(),
    aPerformance.audience(),
    calculator.amount(),
    volumeCreditsFor.apply(aPerformance));
}
```

Moving Functions into the Calculator

```
 Function<Performance,Integer> volumeCreditsFor = aPerformance -> {
   var result = 0;
   result += Math.max(aPerformance.audience() - 30, 0);
   if ("comedy" == playFor.apply(aPerformance).type())
     result += Math.floor(aPerformance.audience() / 5);
   return result;
};
```

```
Function<Performance,EnrichedPerformance> enrichPerformance = aPerformance -> {
  var calculator = new PerformanceCalculator(aPerformance, playFor.apply(aPerformance));
  return new EnrichedPerformance(
    aPerformance.playID(),
    calculator.play(),
    aPerformance.audience(),
    calculator.amount(),
    volumeCreditsFor.apply(aPerformance));
}
```

Martin Fowler

@martinfowler

I repeat the same process to move the **volume credits calculation**.

the process seen in the previous three slides.

Move Function volumeCreditsFor

1. (copy logic over to new context)
2. (delegation)

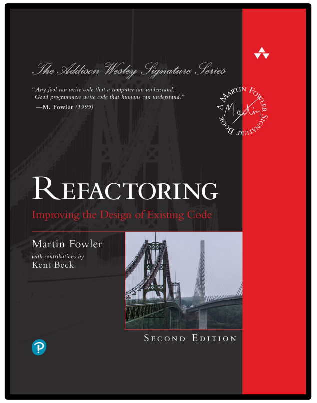
Inline Function volumeCreditsFor

```
record PerformanceCalculator(Performance performance, Play play) {
  int amount() {
    …
  }
  int volumeCredits() {
    var result = 0;
    result += Math.max(performance.audience() - 30, 0);
    if ("comedy" == play.type())
      result += Math.floor(performance.audience() / 5);
    return result;
  }
}
```

```
Function<Performance,EnrichedPerformance> enrichPerformance = aPerformance -> {
  var calculator = new PerformanceCalculator(aPerformance, playFor.apply(aPerformance));
  return new EnrichedPerformance(
    aPerformance.playID(),
    calculator.play(),
    aPerformance.audience(),
    calculator.amount(),
    calculator.volumeCredits());
}
```

Martin Fowler
@martinfowler

- Decomposing the `statement` Function
  - Removing the play Variable
  - Extracting Volume Credits
  - Removing the `formatter` Variable
  - Removing Total Volume Credits
  - Removing Total Amount
  - Status: Lots of Nested Functions
  - Splitting the Phases of Calculation and Formatting
  - Status: Separated into Two Files (and Phases)
  - Reorganising the Calculations by Type
    - Creating a Performance Calculator
    - Moving Functions into the Calculator
    - Making the Performance Calculator Polymorphic



Making the Performance Calculator Polymorphic

```
record PerformanceCalculator(Performance performance, Play play) {
 int amount() {
  var result = 0;
  switch (play.type()) {
   case "tragedy" -> {
    result = 40_000;
    if (performance.audience() > 30)
     result += 1_000 * (performance.audience() - 30); }
   case "comedy" -> {
    result = 30_000;
    if (performance.audience() > 20)
     result += 10_000 + 500 * (performance.audience() - 20);
    result += 300 * performance.audience(); }
   default -> throw new IllegalArgumentException(
      "unknown type " + play.type());
  }
  return result;
 }
 int volumeCredits() {
  var result = 0;
  result += Math.max(performance.audience() - 30, 0);
  if ("comedy" == play.type())
   result += Math.floor(performance.audience() / 5);
  return result;
 }
}
```

```
sealed interface PerformanceCalculator {
 Performance performance();
 Play play();
 default int amount() {
  var result = 0;
  switch (play().type()) {
   case "tragedy" -> {
    result = 40_000;
    if (performance().audience() > 30)
     result += 1_000 * (performance().audience() - 30); }
   case "comedy" -> {
    result = 30_000;
    if (performance().audience() > 20)
     result += 10_000 + 500 * (performance().audience() - 20);
    result += 300 * performance().audience(); }
   default -> throw new IllegalArgumentException(
      "unknown type " + play().type());
  }
  return result;
 }
 default int volumeCredits() {
  var result = 0;
  result += Math.max(performance().audience() - 30, 0);
  if ("comedy" == play().type())
   result += Math.floor(performance().audience() / 5);
  return result;
 }
 static PerformanceCalculator instance(Performance aPerformance, Play aPlay) {
  return switch (aPlay.type()) {
   case "tragedy" -> new TragedyCalculator(aPerformance, aPlay);
   case "comedy" -> new ComedyCalculator(aPerformance, aPlay);
   default -> throw new IllegalArgumentException(
    String.format("unknown type '%s'", aPlay.type()));
  };
 }
}

record TragedyCalculator(Performance performance, Play play) implements PerformanceCalculator { }
record ComedyCalculator(Performance performance, Play play) implements PerformanceCalculator { }
```

Now that I have the logic in a class, it's time to apply the polymorphism. The first step is to use Replace Type Code with Subclasses to introduce subclasses instead of the type code.

Martin Fowler

@martinfowler

In Java, we decided to map the superclass to an interface, and the subclasses to implementations of the interface.

```
Function<Performance,EnrichedPerformance> enrichPerformance = aPerformance -> {
  final var calculator = new PerformanceCalculator(aPerformance, playFor.apply(aPerformance));
  return new EnrichedPerformance(
    aPerformance.playID(),
    calculator.play(),
    aPerformance.audience(),
    calculator.amount(),
    calculator.volumeCredits());
}
```

To make the changes on the previous slide work we also have to make this simple change.

```
Function<Performance,EnrichedPerformance> enrichPerformance = aPerformance -> {
  final var calculator = PerformanceCalculator.instance(aPerformance, playFor.apply(aPerformance));
  return new EnrichedPerformance(
    aPerformance.playID(),
    calculator.play(),
    aPerformance.audience(),
    calculator.amount(),
    calculator.volumeCredits());
}
```

Making the Performance Calculator Polymorphic

```
sealed interface PerformanceCalculator {
  Performance performance();
  Play play();
  default int amount() {
    var result = 0;
    switch (play().type()) {
      case "tragedy" -> {
        result = 40_000;
        if (performance().audience() > 30)
          result += 1_000 * (performance().audience() - 30); }
      case "comedy" -> {
        result = 30_000;
        if (performance().audience() > 20)
          result += 10_000 + 500 * (performance().audience() - 20);
        result += 300 * performance().audience(); }
      default -> throw new IllegalArgumentException(
            "unknown type " + play().type());
    }
    return result;
  }
  default int volumeCredits() {
    var result = 0;
    result += Math.max(performance().audience() - 30, 0);
    if ("comedy" == play().type())
      result += Math.floor(performance().audience() / 5);
    return result;
  }
  static PerformanceCalculator instance(Performance aPerformance, Play aPlay) {
    …
  }
}

record TragedyCalculator(Performance performance, Play play)
implements PerformanceCalculator { }
record ComedyCalculator(Performance performance, Play play)
implements PerformanceCalculator { }
```

This sets up the **structure** for the **polymorphism**, so I can now move on to **Replace Conditional with Polymorphism**.

Martin Fowler

```
sealed interface PerformanceCalculator {
  Performance performance();
  Play play();
  default int amount() {
    var result = 0;
    switch (play().type()) {
      case "tragedy" -> throw new IllegalArgumentException("bad thing");
      case "comedy" -> {
        result = 30_000;
        if (performance().audience() > 20)
          result += 10_000 + 500 * (performance().audience() - 20);
        result += 300 * performance().audience(); }
      default -> throw new IllegalArgumentException(
            "unknown type " + play().type());
    }
    return result;
  }
  default int volumeCredits() {
    var result = 0;
    result += Math.max(performance().audience() - 30, 0);
    if ("comedy" == play().type())
      result += Math.floor(performance().audience() / 5);
    return result;
  }
  static PerformanceCalculator instance(Performance aPerformance, Play aPlay) {
    …
  }
}
record TragedyCalculator(Performance performance, Play play)
implements PerformanceCalculator {
  @Override public int amount() {
    var result = 40_000;
    if (performance().audience() > 30)
      result += 1_000 * (performance().audience() - 30);
    return result;
  }
}
record ComedyCalculator(Performance performance, Play play)
implements PerformanceCalculator { }
```

Making the Performance Calculator Polymorphic

```
sealed interface PerformanceCalculator {
 Performance performance();
 Play play();
 default int amount() {
  var result = 0;
  switch (play().type()) {
   case "tragedy" -> throw new IllegalArgumentException("bad thing");
   case "comedy" -> {
    result = 30_000;
    if (performance().audience() > 20)
     result += 10_000 + 500 * (performance().audience() - 20);
    result += 300 * performance().audience(); }
   default -> throw new IllegalArgumentException(
      "unknown type " + play().type());
  }
  return result;
 }
 default int volumeCredits() {
  var result = 0;
  result += Math.max(performance().audience() - 30, 0);
  if ("comedy" == play().type())
   result += Math.floor(performance().audience() / 5);
  return result;
 }
 static PerformanceCalculator instance(Performance aPerformance, Play aPlay) {
  …
 }
}
record TragedyCalculator(Performance performance, Play play)
implements PerformanceCalculator {
 @Override public int amount() {
  var result = 40_000;
  if (performance().audience() > 30)
   result += 1_000 * (performance().audience() - 30); }
  return result;
 }
}
record ComedyCalculator(Performance performance, Play play)
implements PerformanceCalculator { }
```

Now I move the comedy case down too.

Martin Fowler

@martinfowler

```
sealed interface PerformanceCalculator {
 Performance performance();
 Play play();
 int amount();
 default int volumeCredits() {
  var result = 0;
  result += Math.max(performance().audience() - 30, 0);
  if ("comedy" == play().type())
   result += Math.floor(performance().audience() / 5);
  return result;
 }
 static PerformanceCalculator instance(Performance aPerformance, Play aPlay) {
  …
 }
}
record TragedyCalculator(Performance performance, Play play)
implements PerformanceCalculator {
 @Override public int amount() {
  var result = 40_000;
  if (performance().audience() > 30)
   result += 1_000 * (performance().audience() - 30); }
  return result;
 }
}
record ComedyCalculator(Performance performance, Play play)
implements PerformanceCalculator {
 @Override public int amount() {
  var result = 30_000;
  if (performance().audience() > 20)
   result += 10_000 + 500 * (performance().audience() - 20);
  result += 300 * performance().audience();
  return result;
 }
}
```

Making the Performance Calculator Polymorphic

```
sealed interface PerformanceCalculator {
  Performance performance();
  Play play();
  int amount(); {
  default int volumeCredits() {
    var result = 0;
    result += Math.max(performance().audience() - 30, 0);
    if ("comedy" == play().type())
      result += Math.floor(performance().audience() / 5);
    return result;
  }
  static PerformanceCalculator instance(Performance aPerformance, Play aPlay) {
    return switch (aPlay.type()) {
      case "tragedy" -> new TragedyCalculator(aPerformance, aPlay);
      case "comedy" -> new ComedyCalculator(aPerformance, aPlay);
      default -> throw new IllegalArgumentException(
        format("unknown type '%s'", aPlay.type()));
    };
  }
}
record TragedyCalculator(Performance performance, Play play)
implements PerformanceCalculator {
  @Override public int amount() {
    var result = 40_000;
    if (performance().audience() > 30)
      result += 1_000 * (performance().audience() - 30); }
    return result;
  }
}
record ComedyCalculator(Performance performance, Play play)
implements PerformanceCalculator {
  @Override public int amount() {
    var result = 30_000;
    if (performance().audience() > 20)
      result += 10_000 + 500 * (performance().audience() - 20);
    result += 300 * performance().audience();
   return result;
  }
}
```

The next **conditional** to replace is the **volumeCredits calculation**.

Martin Fowler

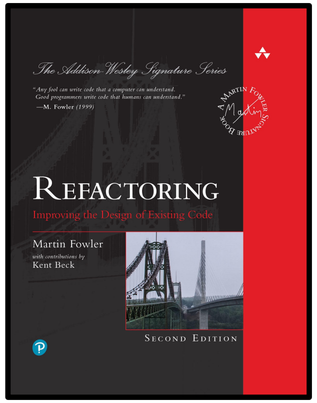
@martinfowler

```
sealed interface PerformanceCalculator {
  Performance performance();
  Play play();
  int amount(); {
  default int volumeCredits() {
    return Math.max(performance().audience() - 30, 0);
  }
  static PerformanceCalculator instance(Performance aPerformance, Play aPlay) {
    return switch (aPlay.type()) {
      case "tragedy" -> new TragedyCalculator(aPerformance, aPlay);
      case "comedy" -> new ComedyCalculator(aPerformance, aPlay);
      default -> throw new IllegalArgumentException(
        format("unknown type '%s'", aPlay.type()));
    };
  }
}
record TragedyCalculator(Performance performance, Play play)
implements PerformanceCalculator {
  @Override public int amount() {
    var result = 40_000;
    if (performance().audience() > 30)
      result += 1_000 * (performance().audience() - 30); }
    return result;
  }
}
record ComedyCalculator(Performance performance, Play play)
implements PerformanceCalculator {
  @Override public int amount() {
    var result = 30_000;
    if (performance().audience() > 20)
      result += 10_000 + 500 * (performance().audience() - 20);
    result += 300 * performance().audience();
   return result;
  }
  @Override public int volumeCredits() {
    return PerformanceCalculator.super.volumeCredits()
      + (int) Math.floor(performance().audience() / 5);
  }
}
```

Martin Fowler

@martinfowler

- Decomposing the `statement` Function
  - Removing the play Variable
  - Extracting Volume Credits
  - Removing the `formatter` Variable
  - Removing Total Volume Credits
  - Removing Total Amount
  - Status: Lots of Nested Functions
  - Splitting the Phases of Calculation and Formatting
  - Status: Separated into Two Files (and Phases)
  - Reorganising the Calculations by Type
    - Creating a Performance Calculator
    - Moving Functions into the Calculator
    - Making the Performance Calculator Polymorphic
  - Status: Creating the Data with the Polymorphic Calculator


The Addison-Wesley Signature Series
Refactoring
Improving the Design of Existing Code
Martin Fowler
with contributions by
Kent Beck
Second Edition

Martin Fowler
@martinfowler

Time to reflect on what introducing the polymorphic calculator did to the code.

Again, the code has increased in size as I've introduced structure.

The benefit here is that the calculations for each kind of play are grouped together.

If most of the changes will be to this code, it will be helpful to have it clearly separated like this.

Adding a new kind of play requires writing a new subclass and adding it to the creation function.

The example gives some insight as to when using subclasses like this is useful.

See next slide for the initial code.

See the three subsequent slides for the refactored code.

Initial Program

Statement.java

```java
record Performance(String playID, int audience) { }

record Invoice(String customer, List<Performance> performances) { }

record Play(String name, String type) { }

public class Statement {

  static String statement(Invoice invoice, Map<String, Play> plays) {
    …
  }
}
```

```java
static String statement(Invoice invoice, Map<String, Play> plays) {
  var totalAmount = 0;
  var volumeCredits = 0;
  var result = "Statement for " + invoice.customer() + "\n";
  final var formatter = NumberFormat.getCurrencyInstance(Locale.US);
  formatter.setCurrency(Currency.getInstance(Locale.US));
  for(Performance perf : invoice.performances()) {
    final var play = plays.get(perf.playID());
    var thisAmount = 0;
    switch (play.type()) {
      case "tragedy" -> {
        thisAmount = 40_000;
        if (perf.audience() > 30)
          thisAmount +=1_000 * (perf.audience() - 30);
      }
      case "comedy" -> {
        thisAmount = 30_000;
          if (perf.audience() > 20)
            thisAmount += 10_000 + 500 * (perf.audience() - 20);
          thisAmount += 300 * perf.audience();
      }
      default ->
        throw new IllegalArgumentException("unknown type " + play.type());
    }
    // add volume credits
    volumeCredits += Math.max(perf.audience() - 30, 0);
    // add extra credit for every ten comedy attendees
    if ("comedy" == play.type())
      volumeCredits += Math.floor(perf.audience() / 5);

    // print line for this order
    result += "  " + play.name() + ": " + formatter.format(thisAmount/100)
                + " (" + perf.audience() + " seats)\n";
    totalAmount += thisAmount;
  }
  result += "Amount owed is " + formatter.format(totalAmount/100) + "\n";
  result += "You earned " + volumeCredits + " credits\n";
  return result;
}
```

Refactored Program

Domain.java

```java
record Performance(
  String playID,
  int audience
) { }

record EnrichedPerformance(
  String playID,
  Play play,
  int audience,
  Integer amount,
  Integer volumeCredits
) { }

record Invoice(
  String customer,
  List<Performance> performances
) { }

record Play(
  String name,
  String type
) { }

record StatementData(
  String customer,
  List<EnrichedPerformance> performances,
  Integer totalAmount,
  Integer totalVolumeCredits
) { }
```

```java
sealed interface PerformanceCalculator {
  Performance performance();
  Play play();
  int amount(); {
  default int volumeCredits() {
    return Math.max(performance().audience() - 30, 0);
  }
  static PerformanceCalculator instance(Performance aPerformance, Play aPlay) {
    return switch (aPlay.type()) {
      case "tragedy" -> new TragedyCalculator(aPerformance, aPlay);
      case "comedy" -> new ComedyCalculator(aPerformance, aPlay);
      default -> throw new IllegalArgumentException(
        String.format("unknown type '%s'", aPlay.type()));
    };
  }
}
record TragedyCalculator(Performance performance, Play play) implements PerformanceCalculator {
  public int amount() {
    var result = 40_000;
    if (performance().audience() > 30)
      result += 1_000 * (performance().audience() - 30); }
    return result;
  }
}
record ComedyCalculator(Performance performance, Play play) implements PerformanceCalculator {
  public int amount() {
    var result = 30_000;
    if (performance().audience() > 20)
      result += 10_000 + 500 * (performance().audience() - 20);
    result += 300 * performance().audience();
   return result;
  }
  public int volumeCredits() {
    return PerformanceCalculator.super.volumeCredits()
      + (int) Math.floor(performance().audience() / 5);
  }
}
```

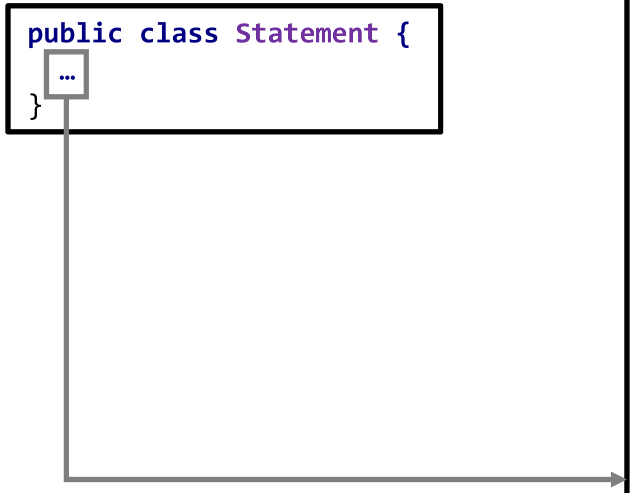
Refactored Program

CreateStatementData.java

```java
public class CreateStatementData {

  static StatementData createStatementData(Invoice invoice, Map<String, Play> plays) {

    Function<Performance,Play> playFor = aPerformance -> plays.get(aPerformance.playID());

    Function<List<EnrichedPerformance >,Integer> totalVolumeCredits = performances ->
      performances.stream().collect(
        reducing(0,EnrichedPerformance::volumeCredits,Integer::sum));

    Function<List<EnrichedPerformance>,Integer> totalAmount = performances ->
      performances.stream().collect(
        reducing(0,EnrichedPerformance::amount,Integer::sum));

    Function<Performance,EnrichedPerformance> enrichPerformance = aPerformance -> {
      final var calculator = PerformanceCalculator.instance(aPerformance, playFor.apply(aPerformance));
      return new EnrichedPerformance(
        aPerformance.playID(),
        calculator.play(),
        aPerformance.audience(),
        calculator.amount(),
        calculator.volumeCredits());
    }

    final var enrichedPerformances =
     invoice.performances().stream().map(enrichPerformance).collect(toList());

    return new StatementData(
      invoice.customer(),
      enrichedPerformances,
      totalAmount.apply(enrichedPerformances),
      totalVolumeCredits.apply(enrichedPerformances));
  }

}
```

Refactored Program

Statement.java



```
public class Statement {
  …
}
```

```
static String statement(Invoice invoice, Map<String, Play> plays) {
  return renderPlainText(createStatementData(invoice,plays));
}

static String htmlStatement(Invoice invoice, Map<String, Play> plays) {
  return renderHtml(CreateStatementData.createStatementData(invoice, plays));
}

static String renderPlainText(StatementData data) {
  var result = "Statement for " + data.customer() + "\n";
  for(EnrichedPerformance perf : data.performances())
    result += "  "+ perf.play().name()+": "+usd(perf.amount()/100)
                   + " (" + perf.audience() + " seats)\n";
  result += "Amount owed is " + usd(data.totalAmount()/100) + "\n";
  result += "You earned " + data.totalVolumeCredits() + " credits\n";
  return result;
}

static String renderHtml(StatementData data) {
  var result = "<h1>Statement for ” + data.customer() + "</h1>\n";
  result += "<table>\n";
  result += "<tr><th>play</th><th>seats</th><th>cost</th></tr>\n";
  for (EnrichedPerformance perf : data.performances()) {
    result += "<tr><td>" + perf.play().name() + "</td><td>" + perf.audience() + "</td>";
    result += "<td>" + usd(perf.amount() / 100) + "</td></tr>\n";
  }
  result += "</table>\n";
  result += "<p>Amount owed is <em>" + usd(data.totalAmount()/100) + "</em></p>\n";
  result += "<p>You earned <em>” + data.totalVolumeCredits() + "</em> credits</p>\n";
  return result;
}

static String usd(int aNumber) {
  final var formatter = NumberFormat.getCurrencyInstance(Locale.US);
  formatter.setCurrency(Currency.getInstance(Locale.US));
  return formatter.format(aNumber);
}
```

To conclude this slide deck, let’s make three more small improvements to the **Java** code.

First, let’s get rid of the remaining **mutability** in the **calculation logic**.

```
sealed interface PerformanceCalculator {
  Performance performance();
  Play play();
  int amount(); {
  default int volumeCredits() {
    return Math.max(performance().audience() - 30, 0);
  }
  static PerformanceCalculator instance(…) {…}
}
record TragedyCalculator(Performance performance, Play play)
implements PerformanceCalculator {
  public int amount() {
    var result = 40_000;
    if (performance().audience() > 30)
      result += 1_000 * (performance().audience() - 30); }
    return result;
  }
}
record ComedyCalculator(Performance performance, Play play)
implements PerformanceCalculator {
  public int amount() {
    var result = 30_000;
    if (performance().audience() > 20)
      result += 10_000 + 500 * (performance().audience() - 20);
    result += 300 * performance().audience();
   return result;
  }
  public int volumeCredits() {
    return PerformanceCalculator.super.volumeCredits()
      + Math.floor(performance().audience() / 5);
  }
}
```

```
sealed interface PerformanceCalculator {
  Performance performance();
  Play play();
  int amount(); {
  default int volumeCredits() {
    return Math.max(performance().audience() - 30, 0);
  }
  static PerformanceCalculator instance(…) {…}
}
record TragedyCalculator(Performance performance, Play play)
implements PerformanceCalculator {
  public int amount() {
    final var basicAmount = 40_000;
    final var largeAudiencePremiumAmount =
      performance.audience() <= 30 ? 0 : 1_000 * (performance.audience() - 30);
    return basicAmount + largeAudiencePremiumAmount;
  }
}
record ComedyCalculator(Performance performance, Play play)
implements PerformanceCalculator {
  public int amount() {
    final var basicAmount = 30_000;
    final var largeAudiencePremiumAmount =
      performance.audience() <= 20 ? 0 : 10_000 + 500 * (performance.audience() - 20);
    final var audienceSizeAmount = 300 * performance.audience();
    return basicAmount + largeAudiencePremiumAmount + audienceSizeAmount;
  }
  public int volumeCredits() {
    return PerformanceCalculator.super.volumeCredits()
      + Math.floor(performance().audience() / 5);
  }
}
```

Next, let’s get rid of **mutability** in the **rendering logic**.

```
static String renderPlainText(StatementData data) {
  var result = "Statement for " + data.customer() + "\n";
  for(EnrichedPerformance perf : data.performances())
    result += "  "+ perf.play().name()+": "+usd(perf.amount()/100)
                   + " (" + perf.audience() + " seats)\n";
  result += "Amount owed is " + usd(data.totalAmount()/100) + "\n";
  result += "You earned " + data.totalVolumeCredits() + " credits\n";
  return result;
}
```

```
static String renderPlainText(StatementData data) {
  return
    "Statement for %s\n".formatted(data.customer()) +
      data.performances()
          .stream()
          .map(p ->
              "  %s: %s (%d seats)\n".formatted(
                p.play().name(), usd(p.amount()/100), p.audience())
          ).collect(Collectors.joining()) +
      """
      Amount owed is %s
      You earned %d credits
      """.formatted(usd(data.totalAmount()/100), data.totalVolumeCredits());
}
```

```
static String renderHtml(StatementData data) {
  var result = "<h1>Statement for ” + data.customer() + "</h1>\n";
  result += "<table>\n";
  result += "<tr><th>play</th><th>seats</th><th>cost</th></tr>\n";
  for (EnhancedPerformance perf : data.performances()) {
    result += "<tr><td>" + perf.play().name() + "</td><td>" + perf.audience() + "</td>";
    result += "<td>" + usd(perf.amount() / 100) + "</td></tr>\n";
  }
  result += "</table>\n";
  result += "<p>Amount owed is <em>" + usd(data.totalAmount()/100) + "</em></p>\n";
  result += "<p>You earned <em>” + data.totalVolumeCredits() + "</em> credits</p>\n";
  return result;
}
```

```
static String renderHtml(StatementData data) {
  return
    """
    <h1>Statement for %s</h1>
    <table>
    <tr><th>play</th><th>seats</th><th>cost</th></tr>
    """.formatted(data.customer()) +
      data
        .performances()
        .stream()
        .map(p -> "<tr><td>%s</td><td>%d</td><td>%s</td></tr>\n"
            .formatted(p.play().name(),p.audience(),usd(p.amount()/100))
        ).collect(Collectors.joining()) +
    """
    </table>
    <p>Amount owed is <em>%s</em></p>
    <p>You earned <em>%d</em> credits</p>
    """.formatted(usd(data.totalAmount()/100), data.totalVolumeCredits());
}
```

@philip_schwarz

And finally, let's make a small change to increase the **readability** of the **totalling functions** for **amount** and **volume credits**.

```
Function<List<EnrichedPerformance>,Integer> totalVolumeCredits = performances ->
  performances.stream().collect(
    reducing(0,EnrichedPerformance::volumeCredits,Integer::sum));

Function<List<EnrichedPerformance>,Integer> totalAmount = performances ->
  performances.stream().collect(
    reducing(0,EnrichedPerformance::amount,Integer::sum));
```

```
Function<List<EnrichedPerformance>,Integer> totalVolumeCredits = performances ->
  performances.stream().mapToInt(EnrichedPerformance::volumeCredits).sum();

Function<List<EnrichedPerformance>,Integer> totalAmount = performances ->
  performances.stream().mapToInt(EnrichedPerformance::amount).sum();
```

That’s all.

I hope you found it useful.